

# hitb magazine

KEEPING KNOWLEDGE FREE

Volume 1, Issue 4, October 2010 [magazine.hackinthebox.org](http://magazine.hackinthebox.org)

Stepping Through a  
**MALICIOUS**  
PDF DOCUMENT **32**

Custom Console Hosts on  
**WINDOWS 7** **12**

Cover Story

Notorious Datacenter Support Systems  
Pwning through Outer Sphere **4**

# TEHTRI-Security

Technology-Ethical-Hacker-Trust-Robust-Information-Security

**This is not a game.**  
[www.tehtri-security.com](http://www.tehtri-security.com)

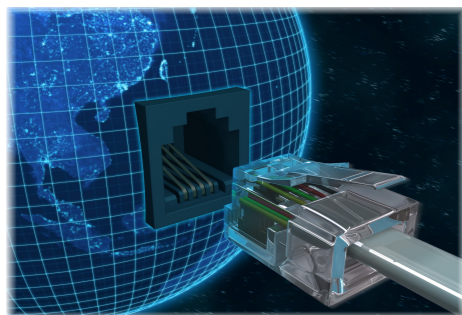
TEHTRI-Security is an innovative French company specialising in Security Consultancy and Expertise for Information and Communication Technology.

### Technical consultancy

- Penetration Tests and Security Audits
- Incident Management, Monitoring...

### Advanced consultancy

- Assistance for countries, companies...
- Fight against information leaks, Protection...



We do know, understand and master the techniques and the methods of attackers (hackers, business intelligence, computer warfare, etc...) as well as the resources needed to counter the current threats.

# hitb

magazine

Volume 1, Issue 4, October 2010

## Editorial

Dear Reader,

Welcome to our fourth issue of 2010! This issue is released in conjunction with HITBSecConf2010 KL. We've had a great first print year and it's all due to you, our loyal readers. Since the first issue back in January, we've seen more than a two-fold readership increase in successive issues. So thank you for your continuing support, and we're excited to bring you this fourth issue which wraps up our 2010 run.

This issue looks at exploitation analysis of help desk systems which is covered by Aditya K. Sood in his article, Notorious Datacenter Support Systems - Pwning through Outer Sphere. We'll also be featuring Decrypting TrueCrypt Volumes with a Physical Memory Dump which shows a simple method to retrieve the volume encryption keys from a memory dump created while the volume was mounted. The author, Jean-Baptiste Bedrune is in fact presenting his talk on Cracking DRM today at HITBSecConf2010 - Kuala Lumpur.

This issue is also bringing back readers' favourite articles from earlier issues - thanks for your feedback through all four issues!

We'll be back again in 2011 with even more cool papers, news and research!

Warmest,

**The Editorial Team**  
[editorial@hackinthebox.org](mailto:editorial@hackinthebox.org)



### Editor-in-Chief

Zarul Shahrin

### Editorial Advisors

Dhillon Andrew Kannabhiran  
 Amy Goh

### Technical Advisor

Gynael Coldwind

### Design

Shamik Kundu  
 (cognitive.designs@gmail.com)

### Website

Bina Kundu

Hack in The Box - Keeping Knowledge Free  
<http://magazine.hackinthebox.org>  
<http://forum.hackinthebox.org>  
<http://conference.hackinthebox.org>

## Contents

**NETWORK SECURITY COVER STORY**  
 Notorious Datacenter Servers Support Systems Pwning Through Outer Sphere **4**

**WINDOWS SECURITY**  
 Custom console hosts on Windows 7 **12**

Windows Objects in Kernel Vulnerability Exploitation **22**

**INFORMATION SECURITY**  
 Stepping Through a Malicious PDF Document **32**

Decrypting TrueCrypt Volumes with a Physical Memory Dump **36**

**MOBILE SECURITY**  
 Reconstructing Dalvik applications using UNDX **44**

**BOOK REVIEW**  
 Ubuntu For Non-Geeks **52**

**INTERVIEW**  
 Aditya Sood **54**

# Notorious Datacenter Support Systems - Pwning through Outer Sphere

## *Exploitation Analysis of Help Desk Systems*

By Aditya K. Sood, SecNiche Security

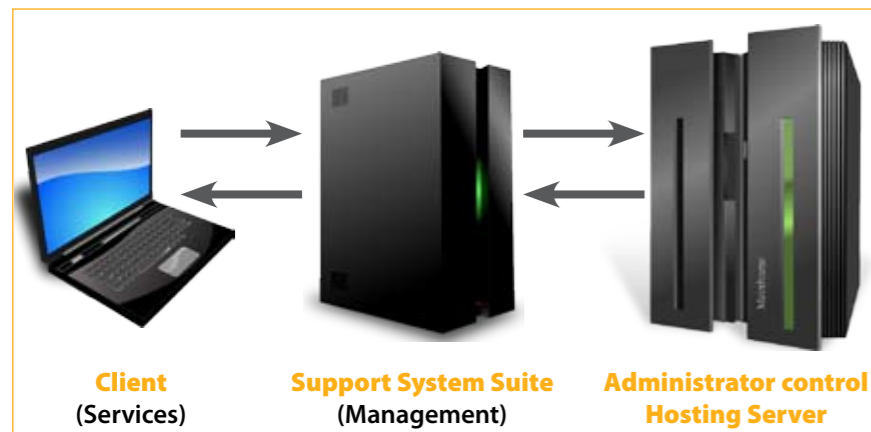
Rohit Bansal, Security Researcher, SecNiche Security

*The online world has been encountering massive levels of malware attacks in the recent times. The outbreak of injected malware has reinforced its devastating stance by contaminating a large number of websites. Most of the traces have been found in the websites under shared and virtual hosting which further includes content from third party delivery networks. Well, it's the truth that a minor inherited weakness in applied software can cause havoc if exploited appropriately. Recent mass level attacks have endorsed this fact. This paper talks about the nature of techniques used by malware writers engaged in performing continuous analysis of differential malware. The paper aims at knowledge sharing by presenting the layout of datacenter compromises through simple support systems used for assisting the customers. The reality of support system shows the nature of insecure work functionality which is exploited heavily by malware writers. This paper is an outcome of real time analysis of compromised systems. This paper has been generalized for security and responsible disclosure reasons.*

**REALITY OF SUPPORT SUITES AND SYSTEMS**

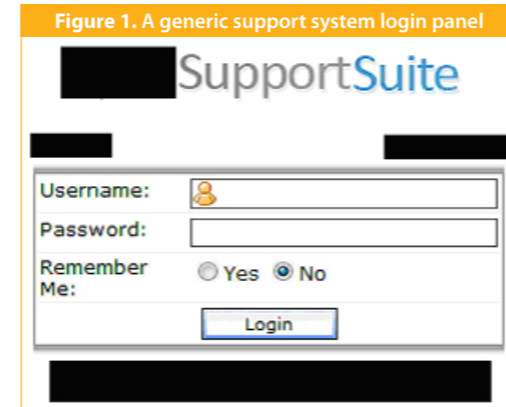
Vulnerabilities always play a critical role in determining the exploitation of an application. It depends a lot on the type of application being compromised and the risk it can pose to the other dependent elements. Hosting service providers and data centers used for client services are being exploited at a large scale in the real time environment. Most of the mass scale attacks have been compromising large datacenters hosting a number of websites on the same servers in production. There are certain specific truths about support suites that are used to manage client's requests which are providing efficient services to them. The understanding can be collaborated as:

- a) The service provider uses centralized support systems to manage clients. It actually utilizes custom designed web application suite which is used to report problems and issues faced by the client while using the services provided by the hosting provider. This is part of good business practice in order to divide technology into different layers and have interface with them individually. Furthermore, any service request issued by the client will go to the support system people who forward the request to the specific administrator in order to resolve the issue. It uses three specific layers as follows
  - a. Client request layer
  - b. Support system management layer
  - c. Administrator request resolving layer
 All these three layers sum up the effectiveness of secured functioning of a hosting provider.



- b) The support usually provides three types of logins as administrator, support and user. All these login accounts have different

set of access rights based on the specific configuration by default. The login panel projects screen as presented in figure 1.



- c) The biggest predicament from human perspective is that the support system people are not very well versed in the principles of security. They are meant only for support by providing an interface layer between user generated requests and the backend administrators to resolve the issues in a timely manner.
- d) Almost all of the supporting suites used a User Ticketing System in order to resolve a user specific request that is actually using services from a specific service provider. Usually, a ticketing system requires a customer to be registered at first in the support system database prior to raising a ticket in the system itself. The customer cannot raise a ticket directly, if the credentials are not registered.

A user issues a ticket to the support system with a unique number for tracking the request. This is an outer sphere of working. The support system verifies the source of information from the customer through an email or direct telephone call in order to confirm the customer's identity.

Once it is done, the support staff administrator or normal support user forwards that request to the specific backend administrator to resolve the issue. A notification is sent as an intermediate step to show the customer that a query has been submitted and is under action. Furthermore, the support system communicates back with the customer once the response is received from the administrator.

In this way, the ticketing system works in the course of supporting suites used for managing servers in data centers. The generic characteristic of support suites is presented in figure 2.

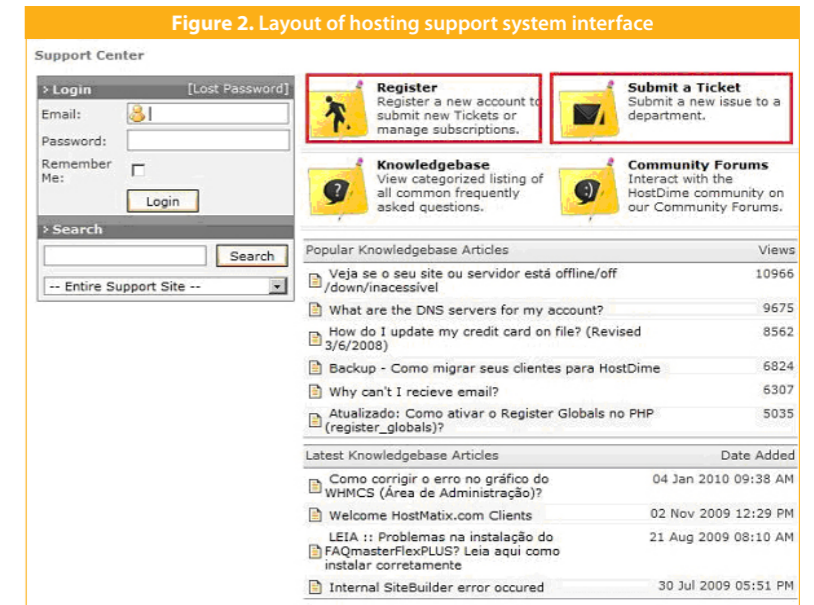
- e) The provision of support of help desk infrastructure on cloud system is also a part of an ongoing process of third party data storage. The databases are hosted on cloud and all specific functions are performed on that basis. The supporting suites have appropriate interface with the Internet as well as the backend servers to provide assistance to the users for resolving the posted queries.

This explains the help desk functionalities and support systems scenario.

**SUPPORT SYSTEM WEAKNESS AND EXPLOITATION – AN ANALYSIS**

The help desk supports suites have a lot of design and inherent issues in the web applications used in real time practice. Our analysis has garnered the artifacts of a number of different techniques that are exploited by the attackers to compromise the supporting suites which will open the door for a large number of user accounts from different websites hosted on the servers present in the data centers. The issues that are exploited in the wild during recent data center compromises are as follows

- a) The Ticketing System is exploited in the wild to leverage the information from different types of vulnerabilities present in the help desk supporting suites. The generic working functionality of the ticketing system has been explained in the last section. The hosting providers allow the customers to be registered directly without any identity checks. Bypassing an identity check is not a large issue but to a certain extent it restricts the control. In the ticketing system, a customer or any user is allowed to register without any stringency after providing a certain set of information. Account credentials are provided to the user after registration which is quite a normal practice. After this process, the customer generates a ticket and submits his query to the supporting staff. Primarily, the supporting staff verifies the identity during that point of time to scrutinize whether the ticket is from the concerned individual or



vice versa. This practice looks appropriate but is not a good design practice in the real environment. The supporting suite itself is a type of web application which works on the same benchmarks as other web applications. The design flaw lies in the fact that after registration the customer is allowed to send tickets directly without any identity check. It is performed afterwards, once the support staff receives it. It provides an edge to the attacker who introduces himself as a customer and is able to send malicious content or stealing links in the assigned tickets. Once the support staff interacts with the ticket or clicks the inserted links, the attack is accomplished. This has been noticed in the recent compromises where the attackers exploit this design bug and further launch web based attacks to exploit the inherent weaknesses in the web based supporting suites. For example, the best choice of attacker is to steal cookies from the supporting suites used by help desk staff.

- b) The second object which enhances the actions for compromising the help desk support systems are inherent vulnerabilities in the web application itself. An attacker requires a XSS weakness in the application itself to combine it with a design bug in the ticketing system to steal the cookies of a particular user in the support staff. Furthermore, the structure of cookie parameters matters whether secure parameters are used or not in order to avoid cookie stealing attacks. There are advanced methods for stealing cookies but







**[WINDOWS SECURITY]**

```
NTSTATUS STDCALL PropertiesDlgShow(HWND hWnd, BOOL SetDefault)
{
    ConsoleState;
    WCHAR SystemDirectory[MAX_PATH];
    UINT DirectoryLength;
    HMODULE hConsoleDll;
    PROP_PROC pfnPropertiesProc;
    NTSTATUS NtStatus;

    GetDefault(&ConsoleState);
    memset(ConsoleState, 0, sizeof(ConsoleState));
    GetSystemDirectory(SystemDirectory, sizeof(SystemDirectory));
    DirectoryLength = sizeof(SystemDirectory);
    if (DirectoryLength <= 0)
        return NtStatus;

    hConsoleDll = LoadLibraryW(SystemDirectory);
    if (hConsoleDll)
    {
        pfnPropertiesProc = GetProcAddress(hConsoleDll, "CPIApplet");
        if (pfnPropertiesProc)
            NtStatus = PropertiesUpdate(&ConsoleState);
    }
}

NTSTATUS PropertiesUpdate(ConsoleState *pConsoleState)
{
    if (pConsoleState)
    {
        if (pConsoleState->hConsoleDll)
        {
            pfnPropertiesProc = GetProcAddress(pConsoleState->hConsoleDll, "CPIApplet");
            if (pfnPropertiesProc)
            {
                NtStatus = pfnPropertiesProc(pConsoleState);
            }
        }
    }
    return NtStatus;
}
```

itself. Instead, it takes advantage of certain system DLL modules, otherwise known as *ServerDlls*. The actual work performed by *CSRSS.exe* is limited to creating a named (*Asynchronous Local Procedure Call port*<sup>3</sup>, loading a few *ServerDlls* (specified in its command-line parameters), calling their initialization routines (e.g. *winsrv.ConServerDllInitialization*), and spawning a dispatcher thread. The latter execution unit is responsible for listening on the (A)LPC port, as well as receiving incoming connections or messages, and passing these to adequate routines, provided by one of the following modules:

- BASESRV.DLL
- WINSRV.DLL
- CSRSRV.DLL

Each *ServerDll* can manage one, or more actual *CsrServers*, whereas a single *CsrServer* is defined by a few characteristics, including:

- The number of supported API routines,
- The first API number supported by the given server,
- A pointer to a - so called - dispatch table, containing pointers of handler routines corresponding to the API functions.

And so, *Table 1* presents a list of the *CsrServers*, assigned to each *ServerDll* listed above, on the Microsoft Vista SP2 (32-bit) operating system. Complete, cross-system (Windows NT4 – Windows 7) lists and tables, presenting names of the functions supported by CSRSS, can be found on the author's blog<sup>4,5</sup>.

CSRSRV.DLL	BASESRV.DLL	WINSRV.DLL
CsrServer	BaseServerApi	ConsoleServer UserServer

Although the developers changed their approach to cross-subsystem support in a relatively early stage of Windows development (by dropping OS/2 after Windows 2000 release), the CSRSS development wasn't abandoned. More specifically, the win32 subsystem has remained an obligatory part of a valid system session. In other words, Windows NT has been unable to complete its tasks without having a CSRSS process running in the background, for all the years of its existence. The above rule is confirmed by system behavior – whenever CSRSS happens to crash – for whatever reason – or is accidentally terminated by a user with adequate privileges, the kernel detects this fact and manually stops the system execution, by triggering a *Blue Screen of Death* (*KeBugCheckEx* routine with the *CRITICAL\_PROCESS\_DIED* parameter). On the other hand, the POSIX (*psxs.exe*) subsystem has also managed to survive, yet belonging to the "optional subsystems" group – it is started on demand, every time a user launches a POSIX application on his desktop.

What should be noted is that the ring-3 CSRSS process was once responsible for performing all of the low-level, GUI related operations in the name of the user's applications. Due to the fact that the user-mode implementation of the graphics services required numerous processor privilege and thread context transitions (i.e. to call native system services and communicate with ring-0 drivers) and thread context transitions, it soon started causing serious efficiency problems, especially in graphics-heavy environments. Although the developers tried their best to optimize both the process – subsystem and subsystem kernel communication channels, the root of the problem still remained. Eventually, the authors decided to directly move the graphics services code into a kernel-mode, under a new name of the *win32k.sys* graphical driver (otherwise known as the ring-0 part of win32 subsystem). Windows NT 4 was the first Microsoft operating system, handling the graphical operations from within the exact

**[WINDOWS SECURITY]**

```
NTSTATUS STDCALL PropertiesDlgShow(HWND hWnd, BOOL SetDefault)
{
    ConsoleState;
    WCHAR SystemDirectory[MAX_PATH];
    UINT DirectoryLength;
    HMODULE hConsoleDll;
    PROP_PROC pfnPropertiesProc;
    NTSTATUS NtStatus;

    GetDefault(&ConsoleState);
    memset(ConsoleState, 0, sizeof(ConsoleState));
    GetSystemDirectory(SystemDirectory, sizeof(SystemDirectory));
    DirectoryLength = sizeof(SystemDirectory);
    if (DirectoryLength <= 0)
        return NtStatus;

    hConsoleDll = LoadLibraryW(SystemDirectory);
    if (hConsoleDll)
    {
        pfnPropertiesProc = GetProcAddress(hConsoleDll, "CPIApplet");
        if (pfnPropertiesProc)
            NtStatus = PropertiesUpdate(&ConsoleState);
    }
}

NTSTATUS PropertiesUpdate(ConsoleState *pConsoleState)
{
    if (pConsoleState)
    {
        if (pConsoleState->hConsoleDll)
        {
            pfnPropertiesProc = GetProcAddress(pConsoleState->hConsoleDll, "CPIApplet");
            if (pfnPropertiesProc)
            {
                NtStatus = pfnPropertiesProc(pConsoleState);
            }
        }
    }
    return NtStatus;
}
```

same level at which the kernel executes – no other major changes have been applied to this architecture, since that time. What actually remained inside CSRSS does not caused efficiency problems anymore, as these APIs have not ever been used too often in regular environments, as opposed to the graphics-related operations.

The console window has been entirely implemented inside one, particular module – that is, *WINSRV.DLL*. The library contains a complete set of handler routines, responsible for performing various, console-related tasks (when requested by the user application). More precisely, a majority of the handlers present inside *ConsoleServerDispatchTable* are basically subsystem-side equivalents of the Windows API functions. *Table 2* presents a few examples of how some of the *kernel32.dll* exports translate into CSR API calls.

WINAPI Function Name	CSRAPI Function Name
<i>kernel32.AllocConsole</i> (exported)	<i>winsrv.SrvAllocConsole</i> (internal)
<i>kernel32.FreeConsole</i> (exported)	<i>winsrv.SrvFreeConsole</i> (internal)
<i>kernel32.GenerateConsoleCtrlEvent</i> (exported)	<i>winsrv.SrvGenerateConsoleCtrlEvent</i> (internal)

All of the messages exchanged between application side modules (*kernel32*, *user32*) and CSRSS *ServerDlls* are sent through the (A)LPC communication channel. The IPC mechanism is, in turn, wrapped by the *ntdll.dll* library – or more precisely – a set of helper routines, such as *CsrClientConnectToServer* or *CsrClientCallServer*. More information about the particular method for exchanging information between client processes and CSRSS is thoroughly described inside the "CSRSS Internals" series<sup>6</sup>.

Our text-based application does not have much of a control over the console window. Instead of being able to send and receive a whole spectrum of supported window

events, the program is limited to a couple of requests, handled by the *WINSRV.DLL* module. Technically, (from the kernel point of view), our process does not have anything in common with the console box in the first place, as CSRSS manages (creates, destroys, dispatches events) the window for us. The above behavior can be easily tested out on any Windows version prior to 7 – it is enough to just grab the console and move it around the desktop as the CSRSS' process CPU usage should immediately increase to several percent, depending on the processor frequency.

Apparently, the described situation does not actually make it easy for us to tweak the console window, due to the fact that a *SYSTEM* privileged process is the owner of "our" window, we are even unable to affect the CSRSS execution, as the security policy will not let us do so (provided our application is running upon an restricted user's right). The circumstances are a little more convenient for users with full administrative rights, as they can at least open the subsystem process and modify its virtual memory contents. By taking advantage of the high user privileges and hooking techniques, one could possibly modify the *WINSRV.DLL* module in-memory, so that the console window behaves in a desired way (e.g. turns invisible on double click).

Another way of altering the appearance or behavior of a console window would require the user to perform a persistent replacement of the *\Windows\system32\winsrv.dll* system file on the hard drive. In such a scenario, any valid PE executable could be used as the new module, as long as it would meet the CSRSS requirements (i.e. valid, exported *CsrServer* initialization routines, correct API handler routines, and more). According to the author, this idea, however, cannot be considered a good choice, because the altering or replacing of critical Windows files on the disk might result in permanent data corruption. Furthermore, the automatic system updates could either

```
NTSTATUS STDCALL PropertiesDlgShow(HWND hWnd, BOOL SetDefault)
```

## WINDOWS SECURITY

```
ConsoleState;
WCHAR SystemDirectory[MAX_PATH];
UINT DirectoryLength;
HMODULE hConsoleDll;
PROP_PROC pfnPropertiesProc;
NTSTATUS NtStatus;

SetDefault);
memset(ConsoleState, 0, sizeof(ConsoleState));
GetConsoleState(&ConsoleState);

DirectoryLength = GetSystemDirectory(SystemDirectory, sizeof(SystemDirectory));
if(DirectoryLength < sizeof(SystemDirectory))
return(NtStatus);

hConsoleDll = LoadLibraryW(SystemDirectory);
if(hConsoleDll)
{
pfnPropertiesProc = GetProcAddress(hConsoleDll, "CPIApplet");
if(pfnPropertiesProc)
return(NtStatus);
}
```

reject the installation on a modified system, or entirely replace the enhanced library, forcing the user to mess with system files over and over again.

Overall, Microsoft made it almost impossible for the user to take more control over the console window, than the original subsystem and security design allows on Windows versions prior to 7. As it turns out, however, the vendor has applied major modifications to the console management design in their latest product, enabling the system users (regardless of their privileges) to take complete control over the console windows associated with the applications of their choice.

### CONSOLE HOSTS ON WINDOWS 7

As presented in the previous chapters, Client/Server Runtime Subsystem was the actual host of the console windows appearing on the user desktop, on regular applications' demand. In fact, all of the window-management logic was implemented in one of the crucial CSRSS modules. From the researcher's point of view is that Inter-Process Communication was being performed every time an old-fashioned program decides to make use of the text interface. What is more, the console support was designed so that it can work with applications running under either high or very low user privileges. And so, in the most extreme scenario, CSRSS had to effectively exchange information with a restricted program with minimal rights. This, in turn, could be used by a local attacker, in order to exploit potential vulnerabilities present in the subsystem process and trigger a code execution in the more privileged application, thus elevating its privileges in the system (into full administrative rights). Not a good scenario, at all.

The concerns of the above nature seem to be justified by events from the past – for example, the MS05-018 advisory<sup>7</sup>, fixing a stack-based buffer overflow vulnerability inside the WINSRV.DLL module, triggered during the

font-name being copied into a local buffer without any length validation. Due to the fact that the vulnerability discoverer claimed the first patch released by Microsoft to be insufficient<sup>8</sup>, a second fix was released after six months of investigation.

In order to address any further issues in the high-privilege console management code, Microsoft made a decision to remove the functionality implementation from the subsystem process, and place it inside a special application, called "Console Host" (conhost.exe). Unlike the Win32 subsystem, the Console host runs in the same security context as the application it is assigned to, so this eliminates any potential *privilege escalation* attacks. In case a security flaw was found in conhost.exe, the attacker would not be able to take advantage of this fact in any useful way. Since every application is making use of the console functionality is assigned its own instance of the conhost.exe process, Denial of Service attacks (i.e. denying console windows for all TUI applications running on the desktop) are not an option, either.

As for the internal, source code-level modifications – only a few relevant changes were actually introduced. Instead of sending numerous LPC requests to the CSRSS process, our application sends one, asking WINSRV.DLL to create a dedicated conhost.exe instance for us. Next then, the application connects to a special port (named, using the following scheme):

```
\\RPC Control\\console-0x%p-lpc-handle
```

with the "%p" format string replaced with the conhost.exe process ID number. From this point on, whenever the application aims to communicate with an external console host, it sends the standard LPC packets to the above port, rather than the Windows Subsystem. *Images 2* and *Images 3* should give you a better understanding of how the

```
NTSTATUS STDCALL PropertiesDlgShow(HWND hWnd, BOOL SetDefault)
```

## WINDOWS SECURITY

```
ConsoleState;
WCHAR SystemDirectory[MAX_PATH];
UINT DirectoryLength;
HMODULE hConsoleDll;
PROP_PROC pfnPropertiesProc;
NTSTATUS NtStatus;

SetDefault);
memset(ConsoleState, 0, sizeof(ConsoleState));
GetConsoleState(&ConsoleState);

DirectoryLength = GetSystemDirectory(SystemDirectory, sizeof(SystemDirectory));
if(DirectoryLength < sizeof(SystemDirectory))
return(NtStatus);

hConsoleDll = LoadLibraryW(SystemDirectory);
if(hConsoleDll)
{
pfnPropertiesProc = GetProcAddress(hConsoleDll, "CPIApplet");
if(pfnPropertiesProc)
return(NtStatus);
}
```

described modifications work in practice.

### BENEFITS

The design reorganization presented in the previous sections supplies the users and researchers with numerous benefits – not only these, publically mentioned by the Microsoft developers. The goal undertaken by these guys is already achieved: by moving yet another part of the CSRSS code into a less-privileged module, the system attack surface has been significantly decreased. For now, this is not what we are actually interested in.

Due to the fact that the security context of the console host has been limited to the current user, restricted TUI applications now have a chance to affect the console host execution path for whatever purpose – such as, tweaking the console appearance on the application's favor. Having free access to the application hosting our console window, one can easily extend it with, theoretically, any functionality he can think of; or better yet – one can even write his own implementation of the default conhost.exe, from stretch!

If we make a step further, it turns out that the Inter-Process communication protocol, implemented by the system conhost.exe executable might be used for purposes other than displaying a console. For instance, the existing LPC communication channel, wrapped with the NTAPI and WINAPI layers, could be utilized by malware, or software protection schemes, in order to make the code logic analysis much harder, and possibly to fool the analyzer himself.

### FEATURES TO BE IMPLEMENTED

Since the Windows users are given new possibilities, it is the right time to take advantage of these. This next section presents a couple ideas of how the existing console box could be modified, so that it becomes more user-friendly during daily routines, or becomes more powerful in its functionality set.

### ANSI ESCAPE CODE

One of the very well known console-related features is the so-called ANSI escape sequences<sup>9</sup>. This functionality makes it possible for applications, relying on text based interaction, to control the overall console box appearance, such as the text-formatting, background and foreground colors, as well as other, platform specific options.

The desired effect (e.g. coloring a particular part of console output) can be achieved, by using special output sequences, which are interpreted by the console in a special manner, rather than just printed on the screen in raw form. As stated by Wikipedia, a great majority of native system consoles running under Linux and other Unix-like systems actively support the escape sequences (and so do external terminal emulators). When it comes to Microsoft products, a special driver called ANSI.SYS existed, being responsible for adding escape-sequences support to the console as was the case for 16-bit console environments (emulated by the aforementioned NTVDM emulator). When it comes to modern, 32-bit Windows applications (such as cmd.exe) making use of console windows, no native escape codes support is provided as the default system terminal just cannot be made to look fancy, by any Microsoft-supported means. On the other hand, a special set of API functions controlling the console appearance is available for the developer<sup>10</sup>, parts of which are presented in *Table 3*.

Apparently, porting Unix-based applications is not a friendly task in the context of console output formatting. Besides, using functions residing in the API layer is not an option for terminal batch in this case – scripts.

Due to the fact that using two-color command line have been considered highly inconvenient (mainly, due to esthetic reasons), several workarounds were implemented

**WINDOWS SECURITY**

```
NTSTATUS STDCALL PropertiesDlgShow(HWND hWnd, BOOL SetDefault)
{
    CONSOLE_STATE ConsoleState;
    WCHAR SystemDirectory[MAX_PATH];
    UINT DirectoryLength;
    HMODULE hConsoleDll;
    PROP_PROC pfnPropertiesProc;
    NTSTATUS NtStatus;

    if (!SetDefault)
        memset(&ConsoleState, 0, sizeof(ConsoleState));
    else
        GetConsoleState(&ConsoleState);

    DirectoryLength = GetSystemDirectory(SystemDirectory, sizeof(SystemDirectory));
    if (DirectoryLength < sizeof(SystemDirectory))
        return NtStatus;

    hConsoleDll = LoadLibraryW(SystemDirectory);
    if (hConsoleDll)
    {
        pfnPropertiesProc = GetProcAddress(hConsoleDll, "CPIApplet");
    }
}
```

**Table 3. Escape sequences' equivalents in the win32 API interface.**

Function name	Comment
SetConsoleTitle	Sets the title for the current console window
SetConsoleTextAttribute	Sets the background and foreground colors of the output text
SetConsoleCursorInfo	Sets the cursor position in the specified console screen buffer

along the way. For instance, Gynvael Coldwind added his own support of the ANSI Escape Codes to *cmd.exe*<sup>11</sup>, by hooking the kernel32.WriteConsoleW import. By taking advantage of the fact that *cmd.exe* uses this function to print every type of console output (including the text echoed by batch scripts), Coldwind was able to recognize the special sequences as they were about to be displayed, and replace these with appropriate calls to the Console API functions. The effect of his work is presented in *Image 4*.

Although such hacks always tended to look very nicely, these solutions have been nothing more but just workarounds – as long as the actual console host remained untouched, it was impossible to achieve a native, system wide escape sequence support. Fortunately, we now have the opportunity to create such mod, by changing the way *conhost.exe* displays characters inside the console box.

**MODIFICATION TECHNICALITIES**

Most of the console modifications are likely to be accomplished, by hooking certain functions, present in the *ConsoleServerApiDispatchTable* array. This table being a straight-forward equivalent of the table from WINSRV.DLL on previous system versions contains most of the functions within our interest. Due to the fact that this is a non-public symbol, one might wonder, how the table address can be actually obtained. Two, most reliable solutions (according to the author) are presented here.

The first easier answer requires the application to recognize the specific version of the *conhost.exe* file, connect to Microsoft servers (provided the computer is connected to the internet) and downloads the appropriate symbol files. Once this is done, our program has access not only to the table address, but the addresses of any other symbol published by Microsoft, as well.

The other solution requires some more knowledge about reverse engineering and Windows architecture. If we take a look at where exactly the *ConsoleServerApiDispatchTable* address is referenced by the *conhost.exe* code, we end up inside a relatively short *ConsoleLpcThread* routine, or more precisely, here:

```
call ds:_ConsoleServerApiDispatchTable[
eax*4]
```

This is due to the fact that the above instruction is the only one meeting the following formatting scheme:

```
call address[reg32*4]
```

in the entire routine, we could basically set a breakpoint at the beginning of *ConsoleLpcThread*, and step over respective instructions in search of the one within our interest (i.e. running our application in the context of the Console Host debugger). In order to find the *ConsoleLpcThread* address, in turn, one could just place an IAT/inline hook on the *CreateThread* import, which is called twice thorough the entire process execution:

1. `CreateThread(NULL, 0, ConsoleLpcThread, NULL, 0, NULL);`
2. `CreateThread(NULL, 0, ConsoleInputThread, NULL, 0, &gdwInputThreadId);`

A very important difference between the two calls from above, is made by the last parameter while being set to

**WINDOWS SECURITY**

```
NTSTATUS STDCALL PropertiesDlgShow(HWND hWnd, BOOL SetDefault)
{
    CONSOLE_STATE ConsoleState;
    WCHAR SystemDirectory[MAX_PATH];
    UINT DirectoryLength;
    HMODULE hConsoleDll;
    PROP_PROC pfnPropertiesProc;
    NTSTATUS NtStatus;

    if (!SetDefault)
        memset(&ConsoleState, 0, sizeof(ConsoleState));
    else
        GetConsoleState(&ConsoleState);

    DirectoryLength = GetSystemDirectory(SystemDirectory, sizeof(SystemDirectory));
    if (DirectoryLength < sizeof(SystemDirectory))
        return NtStatus;

    hConsoleDll = LoadLibraryW(SystemDirectory);
    if (hConsoleDll)
    {
        pfnPropertiesProc = GetProcAddress(hConsoleDll, "CPIApplet");
    }
}
```

NULL while using the "ConsoleLpcThread" pointer, it uses a non-zero value in the other case..

By performing the above steps, one can reliably find the base address of the dispatch table. Thanks to the fact that the API ID numbers do not tend to change between system updates, it becomes possible to replace the existing handlers, for example:

```
dd offset _SrvWriteConsole@8 ;
SrvWriteConsole(x,x)();
if(!SetDefault)
```

with our own implementation of the desired API. Adding the ANSI Escape Code support would rely on forwarding the *SrvWriteConsole* calls to our own stub function, parsing the output text (passed to *conhost* via an LPC request and a shared memory region) and possibly dealing with the escape sequences by calling other *Srv~* routines (like calling *conhost.SrvSetConsoleHostAttribute*) from within the dispatch table (whose address we already know).

Even though Windows 7 has been present on the market for over a year now, the author has not observed any active projects, aiming at enhancing the current console host or re-writing it from the very beginning. Consequently, you as the reader are highly encouraged to be the first one to do it. If you decide to fire up a project of this kind, after reading the article please let me know about it.

**WINDOW TRANSPARENCY**

Another common feature, implemented in most UNIX and External Windows terminals is the transparency setting of the console box the one implemented by the default Console Host does not support this option, though. From the win32 API perspective, manipulating the transparency level of a certain window, is a fairly easy task. In fact, it can be performed with just three lines of C code, as presented in *Listing 1*.

**Listing 1. A code snippet, responsible for setting the transparency degree of a particular window.**

```
BYTE bAlpha = 128; // takes values from the 0..255 range

SetWindowLong(hWnd, GWL_EXSTYLE,
GetWindowLong(hWnd, GWL_EXSTYLE) |
WS_EX_LAYERED);
SetLayeredWindowAttributes(hWnd, 0, bAlpha, LWA_ALPHA);
RedrawWindow(hWnd, NULL, NULL, RDW_ERASE | RDW_INVALIDATE | RDW_FRAME | RDW_ALLCHILDREN);
```

Internally, a few modifications must be applied to *conhost.exe* and possibly other system files – depending on how the user wants to configure the extra appearance settings. Supposedly, the most intuitive choice is to go for the default "Properties" window, fired upon using a context menu option with the same name. What actually happens after doing that, is that a call to an internal *PropertiesDlgShow* function is triggered, which is fully responsible for displaying the configuration panel, reading the configuration data and applying the settings to the current console window.

The question is what is actually going on, inside the function? As presented in *Listing 2*, the routine tries to import an external library called *console.dll* from the system directory – in case of success, a virtual address of the *CPIApplet* exported symbol is obtained, and called three times (apparently, the console module is implemented as a Control Panel Applet!). During the second call, a well-known dialog box is displayed and starts awaiting user interaction. After the user clicks "OK", all of the graphical controls are read, and their values put into the *ConsoleState* structure. Furthermore, an internal *PropertiesUpdate* routine is called, in order to apply the desired settings, by modifying internal variables and structures.

Diving deeper into the *console.dll* internals, one should find out that the *Properties* window is displayed, using the public *comctl32.PropertySheetW* function. If anyone wanted to extend the default property sheet with additional options, he would need to go through the following steps:

```
NTSTATUS STDCALL PropertiesDlgShow(HWND hWnd, BOOL SetDefault)
```

## WINDOWS SECURITY

```
ConsoleState;
WCHAR SystemDirectory[MAX_PATH];
UINT DirectoryLength;
HMODULE hConsoleDll;
PROP_PROC pfnPropertiesProc;
NTSTATUS NtStatus;

if(SetDefault)
    memset(&ConsoleState,0,sizeof(ConsoleState));
else
    GetConsoleState(&ConsoleState);

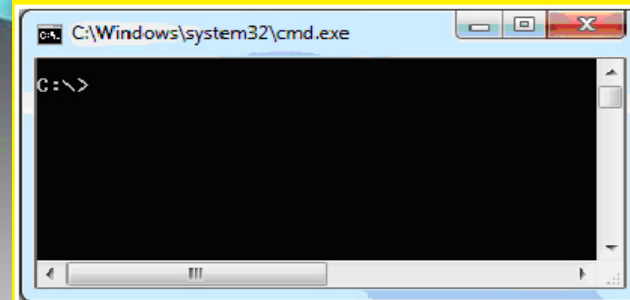
DirectoryLength = GetSystemDirectory(SystemDirectory, sizeof(SystemDirectory));

if(DirectoryLength < sizeof(SystemDirectory))
{
    if(RtlStringCchCatW(SystemDirectory, sizeof(SystemDirectory) - DirectoryLength, L"\\console.dll") >= 0)
    {
        hConsoleDll = LoadLibraryW(SystemDirectory);
        if(hConsoleDll)
        {
            pfnPropertiesProc = GetProcAddress(hConsoleDll, "CPLApplet");
            if(pfnPropertiesProc)
            {
                pfnPropertiesProc(hWnd,1,0,0);
                pfnPropertiesProc(hWnd,5,&ConsoleState,0);
                pfnPropertiesProc(hWnd,7,0,0);
            }
            FreeLibrary(hConsoleDll);
        }
    }
}

NtStatus = LockConsole();
if(!SetDefault)
    NtStatus = PropertiesUpdate(&ConsoleState);

return (NtStatus);
}
```

Image 1. A standard console window on Microsoft Windows Vista.



```
if(pfnPropertiesProc)
{
    pfnPropertiesProc(hWnd,1,0,0);
    pfnPropertiesProc(hWnd,5,&ConsoleState,0);
    pfnPropertiesProc(hWnd,7,0,0);
}
```

Listing 2. A C-like pseudocode of the function called upon using the Properties option from the context menu.

```
NTSTATUS STDCALL PropertiesDlgShow(HWND hWnd, BOOL SetDefault)
{
    CONSOLE_STATE ConsoleState;
    WCHAR SystemDirectory[MAX_PATH];
    UINT DirectoryLength;
    HMODULE hConsoleDll;
    PROP_PROC pfnPropertiesProc;
    NTSTATUS NtStatus;

    if(SetDefault)
        memset(&ConsoleState,0,sizeof(ConsoleState));
    else
        GetConsoleState(&ConsoleState);

    DirectoryLength = GetSystemDirectory(SystemDirectory, sizeof(SystemDirectory));

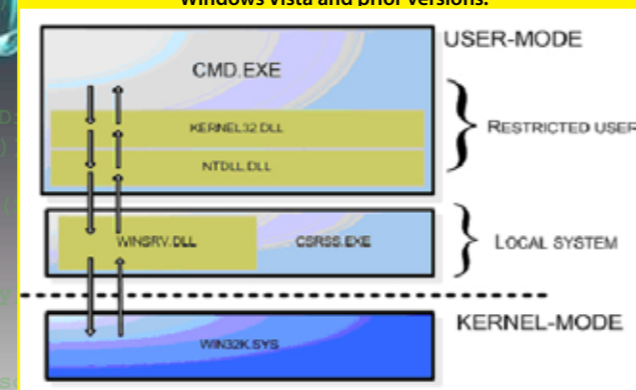
    if(DirectoryLength < sizeof(SystemDirectory))
    {
        if(RtlStringCchCatW(SystemDirectory, sizeof(SystemDirectory) - DirectoryLength, L"\\console.dll") >= 0)
        {
            hConsoleDll = LoadLibraryW(SystemDirectory);
            if(hConsoleDll)
            {
                pfnPropertiesProc = GetProcAddress(hConsoleDll, "CPLApplet");
                if(pfnPropertiesProc)
                {
                    pfnPropertiesProc(hWnd,1,0,0);
                    pfnPropertiesProc(hWnd,5,&ConsoleState,0);
                    pfnPropertiesProc(hWnd,7,0,0);
                }
                FreeLibrary(hConsoleDll);
            }
        }
    }

    NtStatus = LockConsole();
    if(!SetDefault)
        NtStatus = PropertiesUpdate(&ConsoleState);

    return (NtStatus);
}
```

1. Alter the resources (residing in a PE file), describing one of the Properties tabs, adding a new control (e.g. a text edit),
2. Modify the behavior of the dialog box handler routine, assigned to the modified property window, so that it

Image 2. Console management scheme on Windows Vista and prior versions.



actually reads the value of the new control, 3. Think of replacing an existing field inside the CONSOLE\_STATE structure. Due to the fact that it is a fixed-size structure placed on the PropertiesDlgShow's function stack, it is towards impossible to extend it with additional values. And so, one would probably have to change the meaning of a few bytes in the structure, which are now going to store information from the new control, 4. Modify the behavior of PropertiesUpdate, the function called by PropertiesDlgShow after obtaining configuration data from the user. Since one or more CONSOLE\_STATE fields have a different purpose, they must be utilized in a different way, as well (i.e. as an argument to SetLayeredWindowAttributes rather than SetScreenColors (or whatever else)).

Apparently, extending the Properties window with new features is not a very easy task. Fortunately, there is a lot of other options to take – parsing a special .ini configuration file, being one of the easiest one.

### OTHER MODIFICATIONS

It is believed that other, numerous missing functionalities can be found inside the current console window, which might be possibly implemented by interested researchers on Windows 7. What should be noted is that even though the conhost.exe run-time modifications are possible, they might be very hard to apply in a reliable manner. As a basic process running on its own, the executable does not export any symbols – if one wanted to take advantage of this, he could only download them from Microsoft servers; not necessarily a convenient solution.

Due to the above difficulties, creating an alternate version of Microsoft Console Host from the very beginning would be a great choice, in terms of reliability and extensibility.

```
NTSTATUS STDCALL PropertiesDlgShow(HWND hWnd, BOOL SetDefault)
```

## WINDOWS SECURITY

```
CONSOLE_STATE ConsoleState;
WCHAR SystemDirectory[MAX_PATH];
UINT DirectoryLength;
HMODULE hConsoleDll;
PROP_PROC pfnPropertiesProc;
NTSTATUS NtStatus;

if(SetDefault)
    memset(&ConsoleState,0,sizeof(ConsoleState));
else
    GetConsoleState(&ConsoleState);

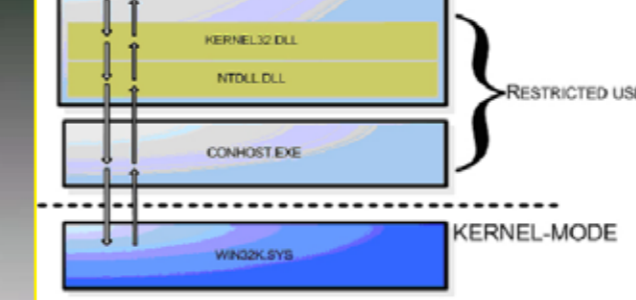
DirectoryLength = GetSystemDirectory(SystemDirectory, sizeof(SystemDirectory));

if(DirectoryLength < sizeof(SystemDirectory))
{
    if(RtlStringCchCatW(SystemDirectory, sizeof(SystemDirectory) - DirectoryLength, L"\\console.dll") >= 0)
    {
        hConsoleDll = LoadLibraryW(SystemDirectory);
        if(hConsoleDll)
        {
            pfnPropertiesProc = GetProcAddress(hConsoleDll, "CPLApplet");
            if(pfnPropertiesProc)
            {
                pfnPropertiesProc(hWnd,1,0,0);
                pfnPropertiesProc(hWnd,5,&ConsoleState,0);
                pfnPropertiesProc(hWnd,7,0,0);
            }
            FreeLibrary(hConsoleDll);
        }
    }
}

NtStatus = LockConsole();
if(!SetDefault)
    NtStatus = PropertiesUpdate(&ConsoleState);

return (NtStatus);
}
```

Image 3. A more secure console box management, introduced in Windows 7.



However, such a project would require enormous amounts of work, especially at the initial stage. A list of major functionalities to be implemented includes:

1. Valid implementation of the Inter-Process communication mechanisms, utilized by conhost.exe and csrss.exe (on previous Windows versions) including appropriate management of the large messages, taking advantages of "Capture Buffers" and a shared heap, winsrv.dll, conhost.exe and client applications, making it possible for regular programs to connect to the console without trouble, and in a secure manner. Furthermore, the custom implementation should not introduce any potential vulnerability to the operating systems, such as allowing low privileged process to connect to the console box requested by the Administrator,
2. Various synchronization mechanisms used by winsrv.dll, conhost.exe and client applications, making it possible for regular programs to connect to the console without trouble, and in a secure manner. Furthermore, the custom implementation should not introduce any potential vulnerability to the operating systems, such as allowing low privileged process to connect to the console box requested by the Administrator,
3. The console box itself is depicted from a graphical point of view. The window drawing procedure would not only need to be super reliable, but also make it easier for the developers to implement additional functionalities related to how the input/output text is being rendered.

All of the above points require thorough knowledge of different parts of the Windows architecture, but once implemented, would be probably made use of for longer usage into the future.

### CONCLUSION

In this paper, the author wanted to present a major design change, introduced in the latest Windows version, as well as show possible ways of taking advantage of this modification on the end-user's favor, rather than keep producing diverse work-around such as "ANSI hack" (being

Image 4. Custom ANSI Escape Code support for cmd.exe.



the only option at the time of its creation). Seemingly, the computer users (i.e. independent researchers) must take care of what the system developers forgot or refused to implement, from time to time, one of the example is the missing console features.

Taking up projects of this kind is not only useful for the overall community, but also tends to expose a lot of the operating system design details, which might come in handy in further work, and provide the researcher with lots of fun during the analysis and development process. Good luck to all of you!

### >>REFERENCES

1. Tavis Ormandy, Microsoft Windows NT #GP Trap Handler Allows Users to Switch Kernel Stack, <http://seclists.org/fulldisclosure/2010/Jan/341>.
2. Jim Martin, Windows 7 / Windows Server 2008 R2: Console Host, <http://blogs.technet.com/b/askperf/archive/2009/10/05/windows-7-windows-server-2008-r2-console-host.aspx>
3. ntdebug, LPC (Local procedure calls) Part 1 architecture, <http://blogs.msdn.com/b/ntdebugging/archive/2007/07/26/lpc-localprocedure-calls-part-1-architecture.aspx>
4. Matthew "j00ru" Jurczyk, Windows CSRSS API List (NT/2000/XP/2003/Vista/2008/7), [http://j00ru.vexillum.org/csrss\\_list/api\\_list.html](http://j00ru.vexillum.org/csrss_list/api_list.html)
5. Matthew "j00ru" Jurczyk, Windows CSRSS API Table (NT/2000/XP/2003/Vista/2008/7), [http://j00ru.vexillum.org/csrss\\_list/api\\_table.html](http://j00ru.vexillum.org/csrss_list/api_table.html)
6. Matthew "j00ru" Jurczyk, Windows CSRSS Write Up: Inter-process Communication (part 2/3), <http://j00ru.vexillum.org/?p=527>
7. Microsoft, Microsoft Security Bulletin MS05-018, <http://www.microsoft.com/technet/security/bulletin/ms05-018.mspx>
8. Cesar Cerrudo, Story of a dumb patch, <http://www.argeniss.com/research/MSBugPaper.pdf>
9. Wikipedia, ANSI escape code, [http://en.wikipedia.org/wiki/ANSI\\_escape\\_code](http://en.wikipedia.org/wiki/ANSI_escape_code)
10. MSDN, Console Functions, <http://msdn.microsoft.com/en-us/library/ms682073%28VS.85%29.aspx>
11. Gynvael Coldwind, Enter teh ANSI Escape Code suport for internal cmd.exe commands and BAT scripts, <http://gynvael.coldwind.pl/?id=130>



# Windows Objects in Kernel Vulnerability Exploitation

By Matthew "j00ru" Jurczyk

Windows kernel vulnerabilities are continuously becoming more and more popular among security experts, in the recent years. This is probably caused by the fact that code running in the mysterious, *ring-0* mode has its own set of rules, as well as potential bugs. Moreover, the possible benefits of exploiting a kernel vulnerability are tremendously different from these, found in user-mode software. Such differences are a simple consequence of the operating system design itself – both processor modes are meant to be used by code responsible for various tasks, such as:

- Security management
- Providing a stable execution environment for user applications
- Physical device management
- Running user-specific programs, such as word processor, internet browser, games etc.

As can be seen, the first three points require considerably higher system privileges, than the latter one. Associating different code modules with different privileges is called *privilege separation*, and is a vital part of *Protected Mode* – the operational mode introduced in the Intel x86 processors in the early 90's. This paper aims to cover some of the possible ways of gathering sensitive data from the Windows kernel, and then using it to elevate the current application privileges, consequently leading to system security compromise.

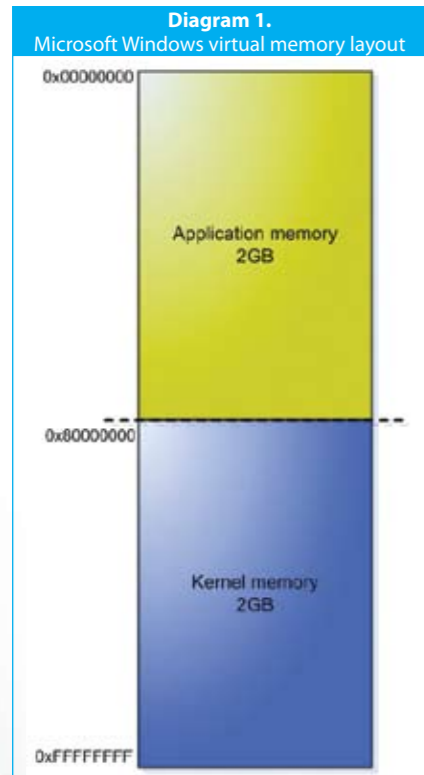


## PROTECTED-MODE BASICS

Before thinking of how the system privileges could be escalated by a potential attacker, one should firstly focus on some basic information about the *Protected Mode* design.

What has been mentioned in the previous section, various system tasks require multiple privilege levels to work on. Thus, in order to provide fair system security, less critical modules should be assigned lower privileges, while the more critical ones should run with full control over the system. To achieve this, Intel introduced four privilege levels (so-called *rings*) - with *ring-0* being the most, and *ring-3* less privileged mode. In practice, most of the modern operating systems only take advantage of *ring-0* and *ring-3*, leaving the remaining two levels unused. Hence, two types of code can be distinguished – *kernel* code (which is not limited to the kernel image, only), having almost complete control over the machine (virtualization mechanisms are beyond the scope of this paper) and *user* code, most commonly executed by ordinary applications, used by the user himself.

One of the most revolutionary features brought by *Pro-*



**Protected Mode** improvements. The general idea, used in Windows until today, is shown in *Diagram 1*. As the image presents, the entire virtual addressing is split into two major parts – *user- and kernel-memory*.

The lower part of the address space is purposed to be accessed by user's applications. As mentioned before, all the programs working on Windows are taking advantage of virtual memory separation – in other words, every single process can operate on his own 2 gigabytes of memory, without sharing it with any other program – this part of memory is *process-specific*. A natural consequence is that user memory is swappable – can be swapped out and saved on the hard disk, when the system is running out of physical memory. Due to the fact that these memory regions are used by non-privileged modules, they can be accessed from within all rings.

The higher part, on the other hand, *belongs* to modules running under *ring-0*. It can be accessed by the system code, only – ordinary applications are unable to execute, modify, or even read its contents. These regions are system-wide, thus don't change on thread switch, but remain the same regardless of the current process. Gaining the ability to execute *ring-0* code makes it possible to subvert the system security, i.e. by installing a stealth *rootkit*, or performing other malicious operations. The entire security design is based on preventing an usual user from altering the existing kernel code or executing his own.

Even though user applications are meant to execute with the *ring-3* rights, a great number of operations cannot be achieved without employing some system management functions, placed in the kernel areas. As noted, it is impossible to directly call privileged code, due to the memory access restrictions. However, a few transition mechanisms have been

developed, allowing *ring-3* to *ring-0* transitioning, such as:

- System calls (SYSENTER/SYSEXIT instructions)
- Interrupts (INT instruction)
- Call Gates (CALL FAR instruction)

All of the above methods let the application call a pre-defined kernel function with a certain number of parameters. In case of syscalls, the system must previously initialize an adequate *Model-specific register* (MSR), interrupts require a valid *Interrupt Descriptor Table* to be present, while Call Gates are based on the *Global/Local Descriptor Table*. As can be seen, all of the methods take advantage of structures managed by the system itself. The user is unable to mess with either GDT or IDT – these structures reside inside kernel memory – or MSR, as the *Write MSR* (WMSR) instruction is reserved for *ring-0* mode.

As shown, probably the only possible way of elevating the security privileges would require finding and exploiting a vulnerability present in a kernel function, that is able to be called by a (potentially hostile) user application.

### THE REAL VALUE OF KERNEL ADDRESSES

Having some elementary knowledge of how Protected Mode works, one could ask about how the kernel addresses could prove useful for a user-mode application, since the process wouldn't be able to access data under that address, after all. On the other hand, numerous vulnerabilities are being found in device drivers, and a majority of them can be classified as *write-what-where conditions*. This particular kind of bug makes it possible to, literally, use the vulnerable driver to write a specified value (**what**) to a chosen location (**where**). Such a situation might be a consequence of many possible scenarios, like lack of input/output pointer sanity checks, *pool-based* buffer overflows, and so on. In order to gain *ring-0* code execution, one must

first choose the appropriate *what* and *where* operands, so that the write operation leads to the desired result.

For the last couple of years, various critical memory locations (playing the *<i>where</i>* role) have been researched and described in detail. This includes places, such as **nt!KiDebugRoutine**<sup>1</sup>, **nt!HalDispatchTable**<sup>2</sup> (exported), **nt!MmUserProbeAddress**<sup>3</sup> (exported), or even the kernel code instructions, themselves! Some of the above methods turned out to be stable and solid, while other remained in the hypothetical state only. One way or another, all of them pose a very interesting subject for further investigation.

### WINDOWS OBJECTS

In order to provide consistent access to various resources made available by the operating system, Windows implements a specific object model. As *Windows Internals 5* states<sup>4</sup>, the object manager (a part of the Windows kernel responsible for object management) was designed to meet the following goals:

- Provide a common, uniform mechanism for using system resources,
- Isolate object protection to one location in the operating system so that C2 security compliance can be achieved,
- Provide a mechanism to charge processes for their use of objects so that limits can be placed on the usage of system resources,
- Establish an object-naming scheme that can readily incorporate existing objects, such as the devices, files, and directions of the file system, or other independent collections of objects,
- Support the requirements of various operating system environments,
- Establish uniform rules for object retention,
- Provide the ability to isolate objects for a specific session to allow for both *local* and *global* objects in the namespace.

In this paper, we are mostly inter-

ested in the *executive* objects, commonly (yet indirectly) utilized by user-mode applications through the Windows API. Some examples of such objects are: files, directories, threads, processes or events. These resources can be tampered with, using functions like *CreateFile*, *WriteFile*, *OpenProcess*, *SetEvent* etc. Each of the above object types represents a certain system resource.

Internally, Windows objects are implemented as basic structures, containing *type-specific* information. Since these structures are stored inside kernel memory, and thus no application has direct access to its contents, all the desired operations are performed by the kernel, on behalf of the user's program. However, *ring-3* code doesn't operate on raw kernel addresses – instead, special values called *Handles* are provided by the Object Manager. These *handles* are actually indexes into the *Process Handle Table*, which in turn contains pointers to the associated structures. In other words, handles are used as the user-mode representatives of system resources, and are translated to real pointers in the kernel mode.

The internal object structure is composed of two integral parts – the object header, common for all existing types of objects, and the latter part – *object-specific* data. The object header includes information such as its name, security descriptor, quota charges and other, standard characteristics. More precisely, it is described by a structure named **OBJECT\_HEADER**, presented in *Listing 1*.

After 24 bytes of the above properties, a next structure follows, depending on the object type. Most of the *executive* object structures are defined in the Microsoft Debugging Symbols<sup>5</sup> for the *ntoskrnl.exe* image. Some exemplary, widely used structure names are: *KPROCESS* (process), *KTHREAD* (thread) or *KSEMAPHORE* (semaphore). More detailed definitions of a few objects are presented later in this paper.

### RETRIEVING OBJECT-RELATED INFORMATION FROM WITHIN USER-MODE

As mentioned before, every single internal object structure is safely stored in the high memory regions, protected from unauthorized write access. Despite that, as it turns out, Windows operating system provides multiple services (system calls), designed to supply a variety of information regarding the current system state. A list of the most important information-querying functions follows:

- **NtQuerySystemInformation**<sup>6</sup> – returns system-wide information, such as kernel configuration (e.g. *memory pools*), hardware information (e.g. processor characteristics), global system settings (e.g. current time), and much more,
- **NtQueryInformationProcess**<sup>7</sup> – returns information about a certain process, based on internal process structures like *KPROCESS*,
- **NtQueryInformationThread**<sup>8</sup> – same as above, involving the thread object,
- **NtQueryJobObject**, **NtQueryInformationToken**, **NtQueryInformationPort** and other – return *type-*

```

Listing 1. Definition of the OBJECT_HEADER structure on Windows 7 RC x86
nt!_OBJECT_HEADER
+0x000 PointerCount      : Int4B
+0x004 HandleCount      : Int4B
+0x008 NextToFree       : Ptr32 Void
+0x00c Lock              : EX_PUSH_LOCK
+0x00d TypeIndex        : UChar
+0x00e TraceFlags       : UChar
+0x00f InfoMask         : UChar
+0x010 Flags            : UChar
+0x011 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
+0x014 QuotaBlockCharged : Ptr32 Void
+0x018 SecurityDescriptor : Ptr32 Void
+0x01c Body             : _QUAD
    
```



Listing 2. Definitions of the structures return by the NtQuerySystemInformation system call

```
typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO {
    USHORT UniqueProcessId;
    USHORT CreatorBackTraceIndex;
    UCHAR ObjectTypeIndex;
    UCHAR HandleAttributes;
    USHORT HandleValue;
    PVOID Object;
    ULONG GrantedAccess;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO, *PSYSTEM_HANDLE_TABLE_ENTRY_INFO;

typedef struct _SYSTEM_HANDLE_INFORMATION {
    ULONG NumberOfHandles;
    SYSTEM_HANDLE_TABLE_ENTRY_INFO Handles[ 1 ];
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;

Where:
UniqueProcessId
The Process ID of the owner of the handle.

CreatorBackTraceIndex
Debugging purpose field, usually zero.

ObjectTypeIndex
The object type identifier of the handle in consideration.

HandleAttributes
Contains internal flags, specifying the handle properties (such as PROTECTED_FROM_CLOSE).

HandleValue
The exact handle value, that the owner process is operating on.

Object
The kernel-mode address of the object referred by the handle.

GrantedAccess
Access granted at the time of creating the handle.
```

Listing 3. An exemplary function, retrieving the virtual address of a specified object

```
LPVOID GetHandleAddress(ULONG dwProcessId, USHORT hObject)
{
    NTSTATUS NtStatus;
    SYSTEM_HANDLE_INFORMATION SystemHandle;
    BYTE* HandleInformation;
    DWORD BytesReturned = 0;
    ULONG i;

    NtQuerySystemInformation(SystemHandleInformation,
        &SystemHandle, sizeof (SYSTEM_HANDLE_INFORMATION), &BytesReturned);

    HandleInformation = new BYTE[BytesReturned];
    if (!HandleInformation)
        return NULL;

    if (!NT_SUCCESS(NtQuerySystemInformation(SystemHandleInformation,
        HandleInformation, BytesReturned, &BytesReturned)))
    {
        delete HandleInformation;
        return NULL;
    }

    PSYSTEM_HANDLE_INFORMATION HandleInfo = (typeof(HandleInfo))
    SystemHandleInformation;
    PSYSTEM_HANDLE_TABLE_ENTRY_INFO CurrentHandle = &HandleInfo->
    Handles[0];

    for ( i=0; i<HandleInfo->NumberOfHandles; CurrentHandle++, i++ )
    {
        if (CurrentHandle->UniqueProcessId == dwProcessId &&
            CurrentHandle->HandleValue == (USHORT)hObject)
        {
            LPVOID ReturnAddr = CurrentHandle->Object;
            delete HandleInformation;
            return ReturnAddr;
        }
    }

    delete HandleInformation;
    return NULL;
}
```

specific information about a specific Windows object.

A majority of the *NtQueryInformation~* functions have their counterparts – *NtSetInformation~* – responsible for changing the specified information instead of querying for it. However, among all the available information classes (defined in *ddk\winddk.h* and *ddk\ntapi.h*, can also be found in the *Windows NT 2000 Native API Reference*<sup>9</sup> book), some of them are marked read-only, while other can be changed, as well. Because of the fact that most of the information related to objects is obtained and set using the above routines, they are extensively used by multiple external libraries, such as *kernel32.dll*, which utilize these system calls to implement documented Windows API functions.

The *NtQuerySystemInformation* function along with *SystemHandleInformation* parameter can be used to obtain data regarding all open handles present in the system. On a valid call, the function returns a 32-bit unsigned integer – *NumberOfHandles* – and the appropriate number of *SYSTEM\_HANDLE\_TABLE\_ENTRY\_INFO* structures, each describing a single handle. The definitions of both structures are shown in *Listing 2*.

After successfully reading structures of all the existing system handles, one can easily extract the address of a certain object. The problem is even simpler, when the handle is created in the context of the local process – in this case, both *UniqueProcessId* and *HandleValue* fields are known straight away, which is enough to find the right descriptor structure. *Listing 3* shows an exemplary function, extracting the object structure address based on the two values detailed above.

In practice, one is able to obtain the address of any object, regardless of its type – the only requirement here is that the process in consideration created a handle to the resource, and we

know its numeric value. Being able to find any given object, let's proceed to the next step.

### SOME PARTICULAR WINDOWS OBJECTS IN PRACTICE

In the *Introduction* section of this paper, I mentioned that before exploiting a *write-what-where* vulnerability, one must find a place that – when overwritten – would lead us straight to a privilege elevation. In other words, appropriate fields, such as function pointers, must be found in the object structures to compromise the machine. Additionally, one must be able to get the kernel to use the modified pointer – this, however, doesn't pose a serious problem.

Out of nearly 30 *executive* objects, three objects that illustrate the idea best are described here. These objects are **Timer (KTIMER)**, **Thread (KTHREAD)**, **Process (KPROCESS)**. It is possible to find a few more structures, containing very sensitive fields – keep in mind that overwriting a function pointer is not a necessity. Modifying other, less "ordinal" values could be also a good solution in many cases.

### TIMER OBJECT

The first target on our way to achieve privileged code execution is a **Waitable Timer Object**. As the MSDN documentation states<sup>10</sup>:

*A waitable timer object is a synchronization object whose state is set to signaled when the specified due time arrives. There are two types of waitable timers that can be created: manual-reset and synchronization. A timer of either type can also be a periodic timer.*

This mechanism has been present in Microsoft Windows since the very beginning of NT series, and hasn't changed too much during the past few years. Some of the most important API functions utilized by legitimate user-mode applications, include:

- **CreateWaitableTimer** and **Create-**

Listing 4. The KTIMER structure definition

```
nt!_KTIMER
+0x000 Header : _DISPATCHER_HEADER
+0x010 DueTime : _ULARGE_INTEGER
+0x018 TimerListEntry : _LIST_ENTRY
+0x020 Dpc : Ptr32_KDPC
+0x024 Period : Uint4B
```

Listing 5. The KDPC structure definition

```
nt!_KDPC
+0x000 Type : UChar
+0x001 Importance : UChar
+0x002 Number : Uint2B
+0x004 DpcListEntry : _LIST_ENTRY
+0x00c DeferredRoutine : Ptr32 void
+0x010 DeferredContext : Ptr32 Void
+0x014 SystemArgument1 : Ptr32 Void
+0x018 SystemArgument2 : Ptr32 Void
+0x01c DpcData : Ptr32 Void
```

- **WaitableTimerEx** for creating the object,
- **SetWaitableTimer** for setting the object configuration, such as the interval time, timer period, optional callback routines, and so on. Internally, this function is responsible for the actual modification of the kernel object contents,
- **CancelWaitableTimer** to deactivate the mechanism and **CloseHandle** to entirely give up using the particular object.

Keeping the above names in mind, it's also important to know what system calls are employed while using documented API functions – these are **NtCreateTimer** and **NtOpenTimer** for requesting access to an existing timer or creating one from scratch, **NtSetTimer** for changing the object settings, **NtCancelTimer** for deactivating a chosen timer.

Because of the fact that every Windows object does have its own *type-specific* structure, so have the timers. To be more exact, all the internal timer-management functions operate on a common structure definition – see *Listing 4*.

At a first glance, one might not see any value that could be worth being beneficially overwritten. The important fact, however, is that the DPC acronym stands for *Deferred Procedure Call*, a popular *kernel-mode* Windows

mechanism allowing high-priority task to schedule a procedure to be executed later in time, with lower priority. And so, the *KDPC* structure definition does contain fields that are indeed worth being changed – see *Listing 5*. The pointer to the deferred function is placed inside the *DeferredRoutine* field, found at offset 0x0C (12d).

As shown, having control over the internal *KTIMER* structure would let a potential attacker execute a *ring-0* payload, by forwarding the *Dpc* pointer to the user-mode part of memory, where a new, malicious *KDPC* structure could be easily crafted.

### THREAD OBJECT

The next structure that, after being altered, brings certain benefits, is the structure responsible for storing information about a single thread present in the system. As a relatively complex mechanism, a number of various information regarding every thread must be kept in memory, such as information about *user-* and *ker-*



Listing 7. The *SuspendApc* field initialization

```
PAGELK:0071221D  push  ebx
PAGELK:0071221E  push  ebx
PAGELK:0071221F  push  offset _
KiSuspendThread@12
PAGELK:00712224  push  offset _xHalPrepareForBugcheck@4
PAGELK:00712229  push  offset _KiSuspendNop@20
PAGELK:0071222E  push  ebx
PAGELK:0071222F  push  esi
PAGELK:00712230  lea  eax, [esi+194h]
PAGELK:00712236  push  eax
PAGELK:00712237  call  _KeInitializeApc@32
```

Or, translated into pseudo-code:

```
KeInitializeApc(KTHREAD->SuspendApc, KTHREAD, 0, KiSuspendNop,
xHalPrepareForBugcheck, KiSuspendThread, 0, 0);
```

nel- mode stacks, *Thread Environment Block* pointer, multiple flags, execution priority, processor affinity, and much more. The most interesting part of the *KTHREAD* structure, however, is one specific field called *SuspendApc*, a pointer to the *KAPC* structure. Let's find out what this name stands for!

The APC (*Asynchronous Procedure Call*) mechanism<sup>11</sup> allows system modules to queue a procedure to be called in the context of a chosen thread, either in *ring-3* or *ring-0* mode. Such a procedure is described by the *KAPC* structure which, in turn, is put onto a special *thread-specific* queue. When an appropriate moment comes (i.e. when the thread enters an alerted state, for example by using the *SleepEx*<sup>12</sup> API function), the procedures are called respectively, and their corresponding structures are erased from the queue – most often, until the queue is entirely empty.

The question is – what does it have to do with the *SuspendApc* field in our structure?

Since Windows NT times, a mechanism called *thread suspension* has been supported by the Windows API. This basically means that most threads, belonging to ordinary applications can remain in two, opposite states: active and inactive. In case of the first one, the thread's execution is normally scheduled, based on its affinity, priority, general system state and numerous other factors. In the latter case, however, the thread is considered *frozen* – the operating system doesn't schedule its execution, its current stack contents/processor context doesn't change etc.

Suspending and resuming threads can be achieved by using the *SuspendThread*<sup>13</sup> and *ResumeThread*<sup>14</sup> API functions or, more internally, *NtSuspendThread* along with *NtResumeThread*. The most interesting part of this mechanism is the actual way, of how the execution of an active thread is being suspended after calling an adequate function.

On thread creation, the *KeInitThread* function initializes the *SuspendApc* field with some pre-defined values, which don't change until the thread termination. After that, when an external process decides to suspend our thread, the already-initialized *KAPC* structure is put on the APC queue belonging to the thread in consideration. The *NormalRoutine* function – *KiSuspendThread* in this case – is then immediately called in the context of

the target thread. When the procedure returns, the thread is already suspended. The interesting part of how the mechanisms works is the fact that the user is able to:

1. Retrieve the virtual address of a specified thread's *KTHREAD* structure, and hence the *SuspendApc* field too,
2. Indirectly (through system calls) call the function pointer defined in *KAPC*

If additionally, the user knew a way of overwriting certain kernel memory areas (i.e. using a vulnerable device driver), the *KTHREAD* structure could be successfully utilized in the vulnerability exploitation process.

One thing that should be noted is that using the thread suspension mechanism is being advised against even by Microsoft itself, as it might cause serious stability problem in the context of the application with suspended threads.

The technique covered in this chapter was first described by *skape & Skywing* in the "a catalog of windows local kernel-mode backdoors" article<sup>15</sup>.

**PROCESS OBJECT**

Another object that could be taken into consideration while exploiting a *write-what-where* vulnerability could be the process itself. Just like threads, processes – special *containers* responsible for providing common execution environment (such as memory context) to multiple threads – must also be described by a variety of different parameters. These include kernel / user execution times, thread list, flags, affinity and others. For a complete listing of the *KPROCESS* structure definition, see *Listing 8*.

A variety of fields that could be taken advantage of, can be observed. In this particular case, however, I would like to focus on *LdtDescriptor*.

The Intel x86 architecture supports two types of Descriptor Tables: the

Listing 8. The *KPROCESS* structure definition

```
+0x000 Header : _DISPATCHER_HEADER
+0x010 ProfileListHead : _LIST_ENTRY
+0x018 DirectoryTableBase : UInt4B
+0x01c LdtDescriptor : _KGDTENTRY
+0x024 Int21Descriptor : _KIDTENTRY
+0x02c ThreadListHead : _LIST_ENTRY
+0x034 ProcessLock : UInt4B
+0x038 Affinity : _KAFFINITY_EX
+0x044 ReadyListHead : _LIST_ENTRY
+0x04c SwapListEntry : _SINGLE_LIST_ENTRY
+0x050 ActiveProcessors : _KAFFINITY_EX
+0x05c AutoAlignment : Pos 0, 1 Bit
+0x05c DisableBoost : Pos 1, 1 Bit
+0x05c DisableQuantum : Pos 2, 1 Bit
+0x05c ActiveGroupsMask : Pos 3, 1 Bit
+0x05c ReservedFlags : Pos 4, 28 Bits
+0x05c ProcessFlags : Int4B
+0x060 BasePriority : Char
+0x061 QuantumReset : Char
+0x062 Visited : UChar
+0x063 Unused3 : UChar
+0x064 ThreadSeed : [1] UInt4B
+0x068 IdealNode : [1] UInt2B
+0x06a IdealGlobalNode : UInt2B
+0x06c Flags : _KEXECUTE_OPTIONS
+0x06d Unused1 : UChar
+0x06e IopmOffset : UInt2B
+0x070 Unused4 : UInt4B
+0x074 StackCount : _KSTACK_COUNT
+0x078 ProcessListEntry : _LIST_ENTRY
+0x080 CycleTime : UInt8B
+0x088 KernelTime : UInt4B
+0x08c UserTime : UInt4B
+0x090 VdmTrapHandler : Ptr32 Void
```

Global and Local ones. While GDT is a *per-processor* structure, there can be multiple LDTs available on the system. More precisely, Windows allows at most one LDT to be associated with a single process. Due to the fact that the decision whether to use the local table or not is up to the application itself – it is an optional feature. As a consequence, every process is started without LDT – it can be created and maintained by the system on demand.

The descriptor table management functions are scattered between the Win32 (*kernel32.dll*) and undocumented, native (*ntdll.dll*) API. When one wants to employ the LDT mechanism, he can choose between calling *NtSetInformationProcess* and *NtSetLdtEntries* (both from the Native API set). On the other hand, querying for information about existing descriptors is accomplished by using either *GetThreadSelectorEntry*<sup>16</sup> (Win32 API) or *NtQueryInformationProcess* (Native API).

Because of the volatile nature of LDTs (which have to be changed every time the process context is switched), the system does have to safely store the descriptor, so that it can be copied into GDT when desired, but wouldn't be accessible by the application's code, at the same time – the *KPROCESS* structure seems to be a perfect place for this purpose, and so it is!

As presented in the "GDT and LDT in Windows kernel vulnerability exploitation"<sup>17</sup>, having at least partial control over a segment descriptor may tremendously affect the system security. A potential attacker could try to transform an existing *LDT-type* descriptor into a *ring-0 Call Gate*, or redirect the existing LDT into user-space memory, where further steps would be taken to elevate the execution privileges.

**COMPATIBILITY**

When it comes to kernel-mode exploitation, what counts most is the compatibility across as great number of system

versions, as possible. Let's reflect about whether the techniques presented above, or any other attacks based on overwriting the contents of Windows objects, could be used to develop a stable exploit. The actual exploitation process consists of three major parts: retrieving a certain object's address, preparing data used to overwrite the object, and sending a proper signal to the vulnerable device driver (or modifying the kernel memory by other means).

The presented method of enumerating all handles present in the system – *NtQuerySystemInformation* with the *SystemHandleInformation* parameter is valid for every Windows NT version known by the author, and can be treated as a reliable source of handle-related information. However, obtaining the base address of the object is just the first phase of calculating the virtual address of a particular field. The second part requires a correct offset to be added to the base, which could result in compatibility-related problems. As Microsoft is removing, adding, and changing existing features in both *user-* and *kernel-mode*, the offsets in internal (especially non-documented) structures tend to change very frequently. One possible solution to this problem would be to hardcode offsets from all the *exploit-supported* Windows versions and check the version before performing any WRITE operation in the kernel. Another option would require the attacker to use a *relatively stable* structure, such as *KTIMER*, which hasn't changed since decades.

As for the destination data preparation, the real compatibility depends on the object type of our choice. Although, in most cases, the desired result is having a function pointer modified, and then getting the kernel to call it – in such a situation, no compatibility issues may occur (the function pointer of the attacker's payload doesn't have to be *formed* in any way). The very last part of the actual attack – sending the "launch signal" to the kernel module in consideration -



doesn't pose any problem in the compatibility context.

Taking the above facts into consideration, the only potential, significant issue would regard *object-specific* offsets that could possibly vary from one system version to another – as shown, multiple countermeasures can be taken in order to eliminate this problem. Therefore, methods presented in this paper can be considered relatively stable, in comparison to other, existing techniques.

### CONCLUSION

In this paper, the author wanted to present a general idea of what parts of the Windows kernel could be successfully treated as an attack vector when combined with *extra abilities* (such as overwriting small parts of kernel memory), most often a consequence of a security vulnerability in one of the device drivers.

Out of all the existing possibilities, only three possible attack vectors has been chosen and described

in detail. For sure, a great number of other, interesting (more or less) targets exist – finding and testing them out is left as an exercise for the reader. Furthermore, one could probably find other ways of overwriting the structures covered in this document, e.g. by tampering with other fields. The overall idea, however, remains the same.

Happy vulnerability hunting! •

### >>REFERENCES

1. Ruben Santamarta: *Exploiting Common Flaws In Drivers*, [http://www.reversemode.com/index.php?option=com\\_content&task=view&id=38&Itemid=1](http://www.reversemode.com/index.php?option=com_content&task=view&id=38&Itemid=1)
2. Kostya Kortchinsky: *Real World Kernel Pool Exploitation*, <http://sebug.net/paper/syscanhk/KernelPool.pdf>
3. SoBelt: *How to exploit Windows kernel memory pool*, [http://packetstormsecurity.nl/Xcon2005/Xcon2005\\_SoBelt.pdf](http://packetstormsecurity.nl/Xcon2005/Xcon2005_SoBelt.pdf)
4. Mark Russinovich, David A. Solomon, Alex Ionescu: *Windows Internals 5*
5. Microsoft, Debugging Tools and Symbols
6. Sven B. Schreiber, Tomasz Nowak: *NtQuerySystemInformation*, <http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/System%20Information/NtQuerySystemInformation.html>
7. Sven B. Schreiber, Tomasz Nowak: *NtQueryInformationProcess*, <http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Process/NtQueryInformationProcess.html>
8. Tomasz Nowak: *NtQueryInformationThread*, <http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Thread/NtQueryInformationThread.html>
9. Gary Nebbett: *Windows NT/2000 Native API Reference*
10. MSDN: *Waitable Timer Objects*, [http://msdn.microsoft.com/en-us/library/ms687012\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms687012(VS.85).aspx)
11. MSDN: *Asynchronous Procedure Calls*, [http://msdn.microsoft.com/en-us/library/ms681951\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms681951(VS.85).aspx)
12. MSDN: *SleepEx Function*, [http://msdn.microsoft.com/en-us/library/ms686307\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms686307(VS.85).aspx)
13. MSDN: *SuspendThread Function*, [http://msdn.microsoft.com/en-us/library/ms686345\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms686345(VS.85).aspx)
14. MSDN: *ResumeThread Function*, [http://msdn.microsoft.com/en-us/library/ms685086\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms685086(VS.85).aspx)
15. skape & Skywing: *A Catalog of Windows Local Kernel-mode Backdoor Techniques*, <http://www.uninformed.org/?v=8&a=2&t=sumry>
16. MSDN: *GetThreadSelectorEntry Function*, [http://msdn.microsoft.com/en-us/library/ms679363\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms679363(VS.85).aspx)
17. Matthew „j00ru” Jurczyk, Gynvael Coldwind: *GDT and LDT in Windows kernel vulnerability exploitation*, [vexillum.org/dl.php?call\\_gate\\_exploitation.pdf](http://vexillum.org/dl.php?call_gate_exploitation.pdf)

test drive a mac.  
ask us how

[ i ] store by C-Zone

Lot 2.07, 2nd floor Digital Mall, SS14, Petaling Jaya T - 603.7954.8841 F - 603.7954.8841







# Decrypting TrueCrypt Volumes with a Physical Memory Dump

By Jean-Baptiste Bédrune – SOGETI/ESEC

*TrueCrypt is a popular disk encryption software, running on Windows, Linux and OSX. This article shows a simple method to retrieve the volume encryption keys from a memory dump created while the volume was mounted. It then describes a tool that decrypts a whole volume using these keys. The technique detailed here should work on all Windows versions.*

As explained in the TrueCrypt documentation: "Inherently, unencrypted master keys have to be stored in RAM too. When a non-system TrueCrypt volume is dismounted, TrueCrypt erases its master keys (stored in RAM)". From there, it is obvious that retrieving encryption keys is possible with a dump of physical memory. This attack is out of the scope of the TrueCrypt security model.

For security reasons, memory pages containing encryption keys cannot be swapped on disk. This means that they will always be present in memory. Hence, the technique explained should always work.

**A QUICK BACKGROUND ON TRUENCRYPT**

This part gives the minimum details needed to understand the rest of the article. If you want more details, check the TrueCrypt website, which has a great documentation about the program internals.

**Volume format**

A TrueCrypt volume is a file, that contains the sectors of the encrypted volume. Each volume is mounted using a password, a set of keyfiles, or a token. The volumes are encrypted with AES, Serpent or Twofish using 256 bits key. For increased (?) security, these algorithms can be chained. The mode of operations for the block ciphers is XTS. This mode is adapted to disk encryption; its internals will not be detailed, the only thing to know here is that it needs two keys.

During the volume creation, the user defines one or more encryption algorithms and a hash function. The hash function is used for the key derivation function and the pseudo random number generator.

The TrueCrypt file can contain a "hidden" volume. This hidden volume can be used to store sensitive information: if someone forces you to reveal your password, you give him the password of the "normal" volume. He will not be able to prove that the file also contains a hidden volume, where all your sensitive data resides.

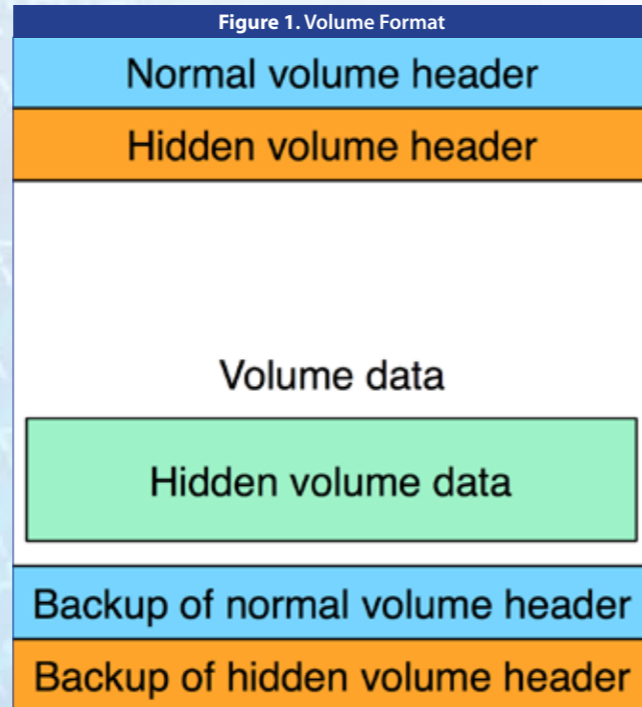
Each file starts with two headers: a header for the normal volume, immediately followed by another one for the hidden volume. In case these headers are altered, for example if a hard disk sector is damaged, another copy of these headers is present at the end of the file.

If the file does not contain a hidden volume, then the hidden volume header is filled with random data, hence there is no way to distinguish it from a real encrypted volume header.

In the other case, the hidden volume is stored *inside* the

normal volume, and not after it so that if somebody mounts a normal volume with its password, he will not be able to see if there is a hidden volume looking at the size of the normal volume. When a normal volume containing a hidden volume is mounted, a legitimate user enters the passwords for the normal *and* the hidden volume; Truecrypt decrypts both headers to compute the size of the normal volume.

**VOLUME SCHEMA**



**Header format**

Each header is 65536 bytes long. Data from offset 512 is filled with random data, and is reserved for future use. Headers contain, among other things, the volume encryption keys so they are obviously encrypted. The only parameter which is not encrypted is a 64 bytes random salt.

Volume headers are decrypted with the password supplied by the user and the random salt. These headers contain:

- The start offset of the encrypted volume (2 x 65536 for a normal volume, just after the two initial headers), and its size.
- Volume encryption keys. 1 to 3 ciphers can be chained, this field contains between 2 and 6 encryption keys.
- Sector size of the volume.
- Data used to control integrity.

**Mounting a volume**

On Windows, TrueCrypt volumes are handled with a filtering driver. A secret is needed to mount a volume. This secret can be either a password, a set of keyfiles, or

a PKCS #11 token. The PKCS #11 token is actually used to store keyfiles; its advantage over a keyfile being a PIN protection feature.

This secret is then copied into a Password structure:

```
#define MAX_PASSWORD 64
// Maximum possible password length

typedef struct
{
    // Modifying this structure can introduce incompatibility with previous versions
    unsigned __int32 Length;
    unsigned char Text[MAX_PASSWORD + 1];
    char Pad[3]; // keep 64-bit alignment
} Password;
```

If the keyfiles are longer than 64 bytes, a derivation algorithm is used to provide a 64 bytes buffer. TrueCrypt creates a `MOUNT_STRUCT` structure containing the Password structure and sends it to its driver with the `TC_IOCTL_MOUNT_VOLUME_IOCTL`.

Hence, all the cryptographic operations needed to mount the volume are done in kernel mode.

Decryption keys are created: the content of the Password structure is derived with the PBKDF2 algorithm using the salt of the volume header. The number of iterations used for PBKDF2 is dependant on the underlying hash function used: 1000 for SHA-512 and Whirlpool, and 2000 for SHA-1. These keys are used to decrypt the volume header, not the volume itself: if a user wants to change its password, only the header has to be updated. That also means that volume encryption keys cannot be changed: if you think your volume has been compromised once, changing the password is a bad idea. Creating a new volume, with new encryption keys, is better.

Then, TrueCrypt tries to decrypt the volume header with the derived keys. Several checks are done on the decrypted header to verify it has been correctly decrypted, i.e. to verify if the password is correct.

- 4 bytes at offset 4 must be the string "TRUE".
- 4 bytes at offsets 72 must be the CRC-32 of the bytes 256 to 511.
- 4 bytes at offset 252 must be the CRC-32 of the bytes 64 to 251.

The driver keeps a context of the encrypted volume. All the data related to cryptographic information is stored in a `CRYPTO_INFO` structure. In this structure reside the

volume encryption keys, obviously needed to perform encryption and decryption operations.

The password is erased from memory, as it is not needed anymore, except if the options "Cache passwords and keyfiles in memory", deactivated by default, is enabled. This means there is generally no way to retrieve it.

**ENCRYPTION KEYS IN MEMORY**

The hypothesis here is that we have obtained a TrueCrypt volume and have taken a snapshot of the physical memory while the volume was mounted. Several possibilities are available to dump the memory like cold boot attacks [COLDBOOT], FireWire [FIREWIRE] or PCI cards [PCI]. This hypothesis is out of the scope of the TrueCrypt security model.

We know that the keys are present in memory as the pages in which they reside are never swapped. One possible way to find them is to rebuild the virtual memory. This can be time consuming and dependant of the operating system version and architecture.

Another quicker way to find them is to do message carving. The idea here is to retrieve the key without rebuilding the virtual memory so that it works independently of the operating system version. The difficulty here is that the keys are random so there is no pattern that will give us their position. One could consider computing the entropy of a memory block and, depending on if it is high or not, consider if it is a possible key. This leads to many false positives, all the probable keys need to be tested.

As said before, data related to cryptography is stored in a `CRYPTO_INFO` structure. Let's look at this structure:

```
typedef struct CRYPTO_INFO_t
{
    int ea;
    /* Encryption algorithm ID */
    int mode;
    /* Mode of operation (e.g., XTS) */
    unsigned __int8 ks[MAX_EXPANDED_KEY];
    /* Primary key schedule (if it is a cascade, it contains multiple concatenated keys) */
    unsigned __int8 ks2[MAX_EXPANDED_KEY];
    /* Secondary key schedule (if cascade, multiple concatenated) for XTS mode. */

    BOOL hiddenVolume;
    /* Indicates whether the volume is mounted/mountable as hidden volume */
};
```

```
#ifndef TC_WINDOWS_BOOT
uint16 HeaderVersion;

GfCtx gf_ctx;

unsigned __int8 master_keydata[MASTER_KEYDATA_SIZE]; /* This holds the volume header area containing concatenated master key(s) and secondary key(s) (XTS mode). For LRW (deprecated/legacy), it contains the tweak key before the master key(s). For CBC (deprecated/legacy), it contains the IV seed before the master key(s). */

unsigned __int8 k2[MASTER_KEYDATA_SIZE]; /* For XTS, this contains the secondary key (if cascade, multiple concatenated). For LRW (deprecated/legacy), it contains the tweak key. For CBC (deprecated/legacy), it contains the IV seed. */

unsigned __int8 salt[PKCS5_SALT_SIZE];
int noIterations;
int pkcs5;
...
}
```

master\_keydata and k2 contain the volume encryption keys. They are both 256 byte buffers. An interesting thing for carving is that, according to the comments, master\_keydata contains both the master and the secondary keys while k2 contains only the secondary keys. Comparing the secondary keys in master\_keydata and k2 gives us a good pattern for carving.

Something more interesting now: the salt used to derive the decryption keys of the volume header is stored just after these keys. Salt is a 64 bytes buffer that contains random data and is stored at the beginning of the TrueCrypt volume file. It is the only information which is public and stored plaintext in the volume file. We now have a very good pattern.

We can add more checks by verifying that noIterations, which is the number of iterations performed during PBKFD2, is 1000 or 2000. Finally, a memory analysis showed that the pkcs5 parameter seems to be always 1.

With all this information, we can certainly retrieve the CRYPTO\_INFO structure easily. Actually, it is not possible to get the whole CRYPTO\_INFO structure if the system has a page size of 4 kB as its size is really bigger, mainly because of the gf\_ctx field used as a workspace for

the Galois field operations of the XTS mode. However, experience has shown that the parameters we are looking for are always on the same page, so this is not a problem.

To find the volume encryption keys, extract its salt from the TrueCrypt volume, and search for it in the memory dump. I chose to check only the noIterations parameters to verify it is really the CRYPTO\_INFO structure that has been found. It worked on all my tests.

Figure 2 shows an extract of the CRYPTO\_INFO structure retrieved from a memory dump.

The 64 bytes seed is highlighted in purple, immediately followed by the number of iterations (0x3E8 = 1000, so the hash used is either SHA-512 or Whirlpool). Above are the master\_keydata and the k2 tables, each of them being 256 bytes long.

The first 128 bytes of master\_keydata are not null, which means it contains 4 AES-256 keys. Remember that the two XTS keys are concatenated in master\_keydata, so here a cascade of two algorithms is used, with:

- Algo1k1 = 10D7BE7DC797FB34248  
124D723BE3D8044C148889CD217022F1F836CACC345
- Algo2k1 = 14E5872E290B688D3AA29153F56D214  
BFD77273D1A229EBC0A05F21246AC6FF4
- Algo1k2 = F7FB16585F814EC48CC3CC9B856A163A4-  
CAD08 B5857B46167039 B79750B29733
- Algo2k2 = D2FC1E546BF79F88076F56FAC25E04  
6B5BA2E6D6094BE85DB9E885E420AF49B6

It can be checked that the secondary keys in the k2 array are the same as the ones in the master\_keydata table.

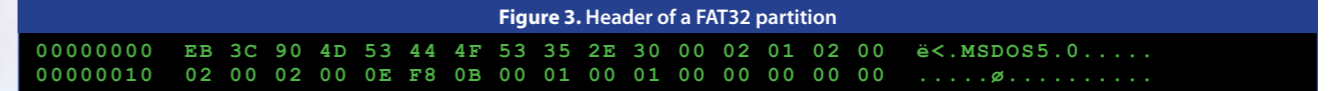
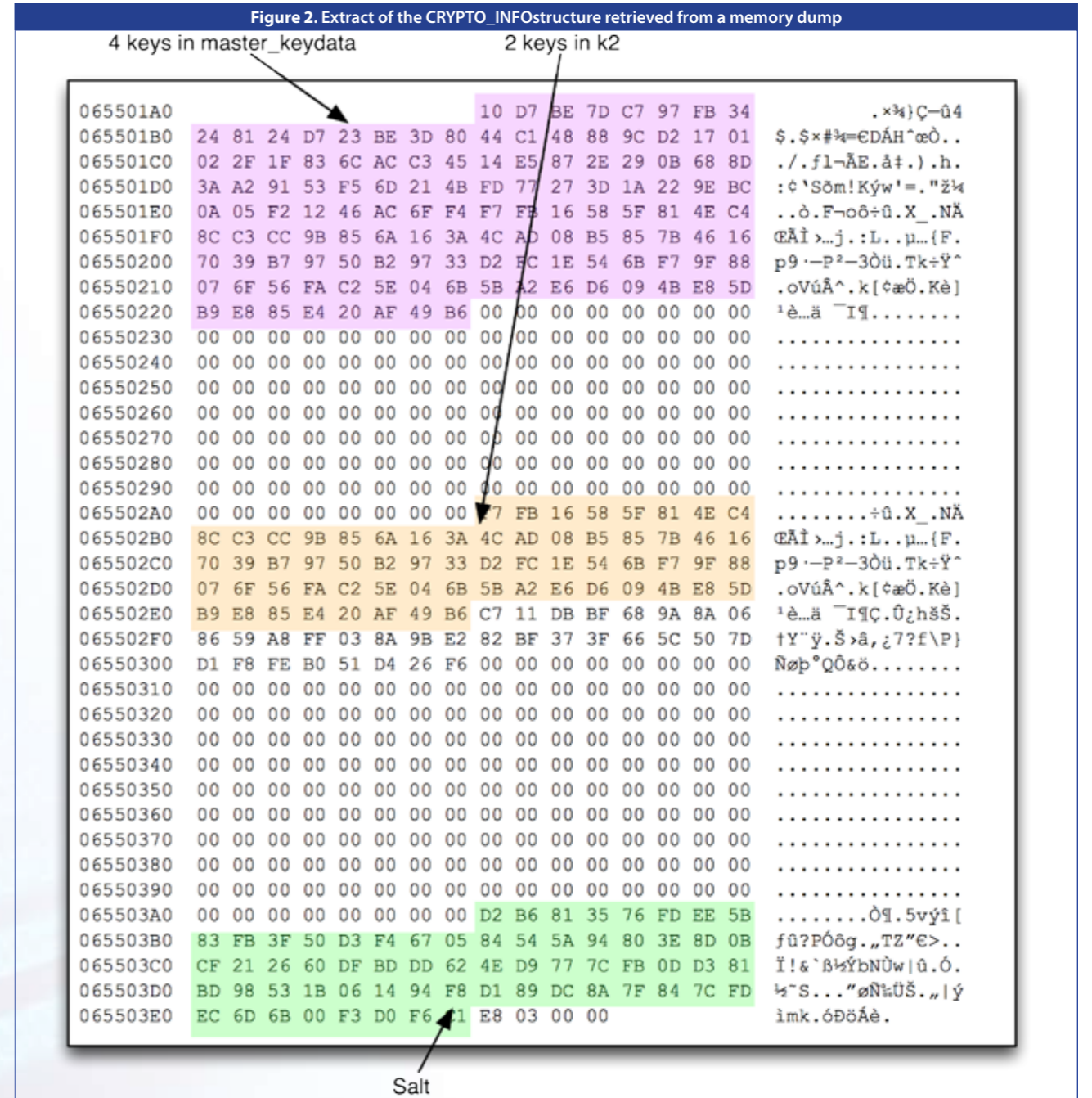
DECRYPTING VOLUMES

Now the keys have been retrieved it is possible to directly decrypt the volume. A problem remains: the encryption algorithms are not known. In the previous example, we only know that a cascade of two algorithms was involved.

How to know which algorithms were used?

The list of the encryption algorithms is not stored in the volume: actually, TrueCrypt tries to decrypt the volume header with all the available algorithms and breaks when the header has been successfully decrypted, i.e when the two CRC-32 values are correct. In our situation, we do not decrypt the volume header, so no integrity check can help us.

When the volume is mounted, the encryption algorithms identifiers are stored in the ea field of the CRYPTO\_INFO structure. Unfortunately, this field is located before the gf\_ctx field, hence it might be present on another page.



The retained solution is to decrypt the first sector of the encryption volume, located just after the volume header, and to check if it is a FAT32 or NTFS volume header. Using master\_keydata, we know how many algorithms are used in cascade. Here are the available algorithms sorted according to the number of cipher involved:

- 1 cipher:
  - o AES
  - o Twofish

- o Serpent
- 2 ciphers:
  - o AES-Twofish
  - o Serpent-AES
  - o Twofish-Serpent
- 3 ciphers:
  - o AES-Twofish-Serpent
  - o Serpent-Twofish-AES

Hence, bruteforce has to be done on at most three possible candidates. A valid header starts with 3B and has an identifier at offset 3 (MSDOS5.0 here).

#### Detecting hidden volumes

Looking at the CRYPTO\_INFO structure gives another interesting results : remember that TrueCrypt can create hidden volumes, and that there is no way to know there is a hidden volume all volumes are dismounted.

This is different when the volume is still mounted.

`volDataAreaOffset` specifies the position of the first data sector of the volume. When a normal volume is mounted, this value is always 0x20000, which is the offset just after the two headers. When a hidden volume is mounted, this value will be different. This characteristic can be used to determine if the retrieved keys are for the normal or the hidden volume.

What is more interesting is that when a user mounts a normal volume that contains a hidden volume, and wants to protect the data in the hidden volume, he enters the two passwords. In this case, the normal volume has the same size as before, but TrueCrypt prevents the hidden volume area to be written. To remember this option, a `bProtectHiddenVolume` flag is set in the `CRYPTO_INFO` structure. This proves the existence of a hidden volume.

#### THE TOOLS

A tool has been developed to retrieve the encryption keys from the memory dump. It searches for the volume salt, checks if it is inside a `CRYPTO_INFO` structure, and dumps the keys.

Another tool decrypts the whole volume using the encryption keys previously retrieved. The goal is to analyze the volume using your favorite forensics tools without mounting it with Windows (volume could be slightly modified, which is bad for a legal forensics analysis).

Finally, a third tool writes a custom volume header, using a chosen password, that contains the encrypted encryption keys. It allows you to mount the volume using a fake password if altering the volume is not important.

Tools could have used the TrueCrypt source code but I preferred to develop them in Python, mainly because of the several tools needed to compile TrueCrypt. The base code comes from a great blog post<sup>1</sup> and has been adapted for TrueCrypt 6/7.

The original code was mainly for learning purposes and was very slow: cryptographic routines were all written in

Python. The worst part was the implementation of the XTS mode that requires computations in  $GF(2^{128})$ ; these computations were not optimized at all, which allowed us to understand how it worked, but made the tool completely unusable in real world.

No crypto library implemented all the hash and encryption primitives used by TrueCrypt. I often use PyCrypto, then I decided to add the missing algorithms: Serpent, Twofish, Whirlpool and SHA-512. The SHA-512 module is a simple wrapper for hashlib; the other algorithms have been written in C using the linux kernel sources.

XTS mode has been written in Python and is way faster than the previous implementation.

#### CONCLUSION

The method shown here is not very technical. Finding keys in memory is rather easy, because of the presence of the volume salt near the encryption key. Nevertheless, it is a useful tool!

Code and PyCrypto patches are available at <http://code.google.com/p/truedecrypt/>.

#### >>BIBLIOGRAPHY

[TCPYTHON] Björn Edström., TrueCrypt explained (TrueCrypt 5 update), 2008  
<http://blog.bjrn.se/2008/02/truedecrypt-explained-truecrypt-5-update.html>

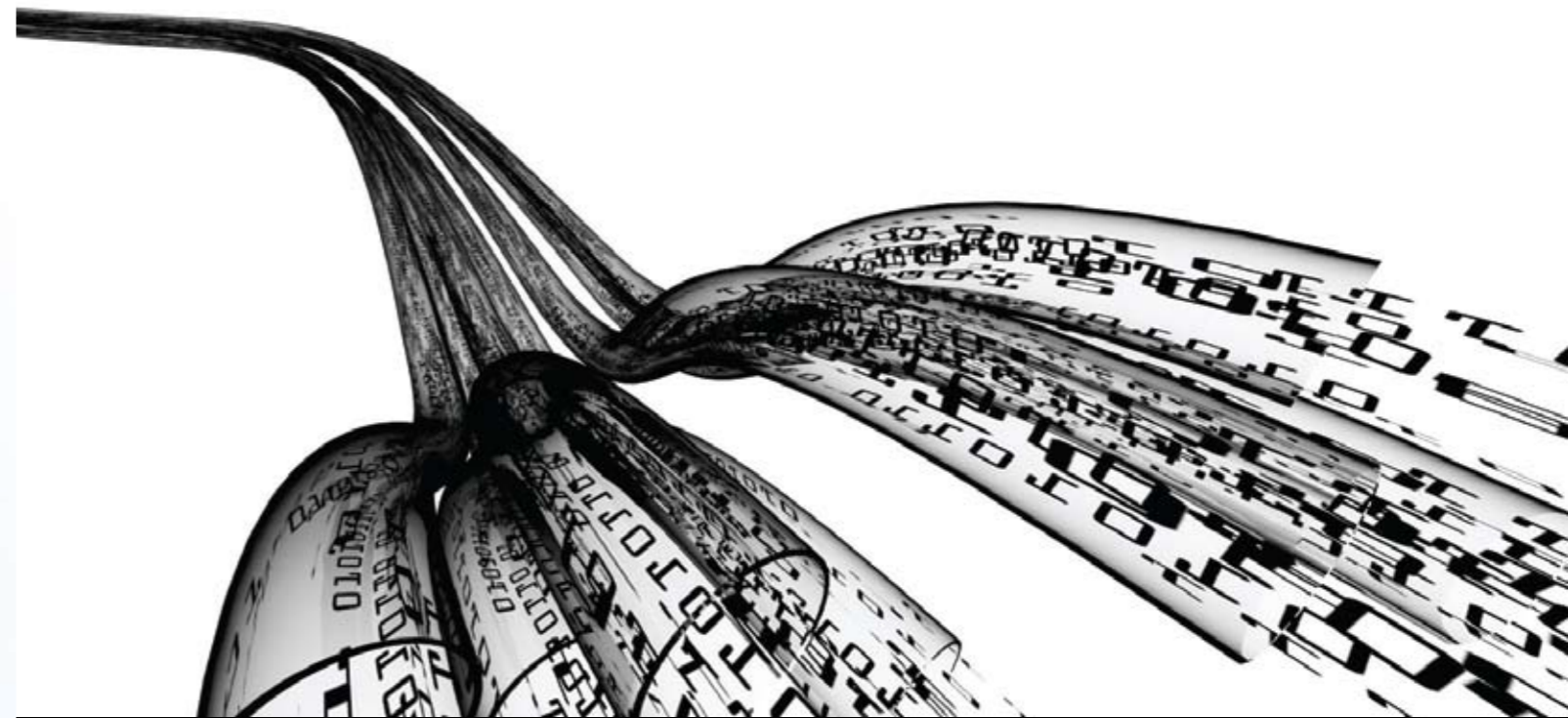
[COLDBOOT] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten (2008-02-21). Lest We Remember: Cold Boot Attacks on Encryption Keys. Princeton University. <http://citp.princeton.edu/memory/>.

[FIREWIRE] D. Aumaitre. A little journey inside Windows memory, 2008, Hack.lu  
<http://esec-lab.sogeti.com/dotclear/public/publications/08-hacklu-memory.pdf>

[PCI] C. Devine, G. Vissian. Compromission physique par le bus PCI, 2009, SSTIC  
[http://actes.sstic.org/SSTIC09/Compromission\\_physique\\_par\\_le\\_bus\\_PCI/SSTIC09-article-C-Devine-G-Vissian-Compromission\\_physique\\_par\\_le\\_bus\\_PCI.pdf](http://actes.sstic.org/SSTIC09/Compromission_physique_par_le_bus_PCI/SSTIC09-article-C-Devine-G-Vissian-Compromission_physique_par_le_bus_PCI.pdf)

#### >>REFERENCES

1. TrueCrypt explained (TrueCrypt 5 update) - <http://blog.bjrn.se/2008/02/truedecrypt-explained-truecrypt-5-update.html>



Harnessing collective knowledge to deliver one-of-a kind capabilities.

Vulnerability Research Labs, LLC combines state-of-the art technology with tradecraft refined by decades of experience, to deliver an unparalleled set of capabilities to mitigate corporate risk posed by today's cyber threats.





# Reconstructing Dalvik Applications using UNDX

By Marc Schönfeld



As a reverse engineer I have the tendency to look in the code that is running on my mobile device. I am coming from a JVM background, so I wanted to know what Dalvik is really about. Additionally I wanted to learn some yet another bytecode language, so Dalvik attracted my attention while sitting on a boring tax form. As I prefer coding to doing boring stuff, I skipped the tax declaration and coded the UNDX tool, which will be presented in the following paragraphs.

## WHAT IS DALVIK

Dalvik is the runtime that runs userspace Android applications. It was invented by Dan Bornstein, a very smart engineer at Google, and he named it after a village in Iceland. Dalvik is register-based and does not run java bytecode. It runs its own bytecode dialect which is executed by this Non-JVM runtime engine, see the comparison in *Table 1*.

	Dalvik	JVM
Architecture	Register	Stack
OS-Support	Android	Multiple
RE-Tools	Few	Many
Executables	APK	JAR
Constant-Pool	per Application	per Class

## DALVIK DEVELOPMENT PROCESS

Dalvik apps are developed using java developer tools on a standard desktop system, like eclipse (see *Figure 1*) or Netbeans IDE. The developer compiles the sources to java classes (as with using the javac tool). In the following step he transform the classes to the dalvik executable format (dx), using the dx tool, which results in the classes.dex file. This file, bundled with meta data (manifest) and media resources form a dalvik application, as a 'apk' deployment unit. An APK-file is transferred to the device or an emulator, which can happen with adb, or in most end-user cases, as download from the android market.



## DALVIK RUNTIME LIBRARIES

A dalvik developer can choose from a wide range of APIs, some known from Java DK, and some are Dalvik specific. Some of the libraries are shown in *Table 2*.

	Dalvik	JVM
java.io	Y	Y
java.net	Y	Y
android.*	Y	N
com.google.*	Y	N
javax.swing.*	N	Y

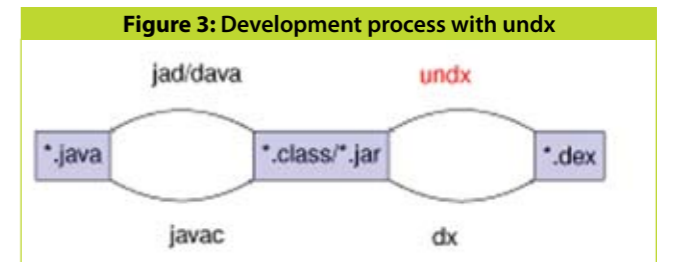
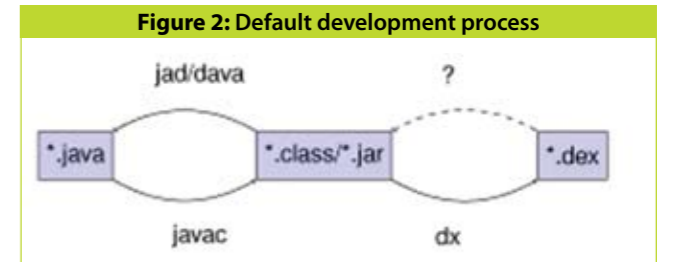
## DALVIK DEVELOPMENT FROM A REVERSE ENGINEERING PERSPECTIVE

### PERSPECTIVES

Dalvik applications are available as apk files, no source included, so you buy/download a cat in the bag. Typical questions during reverse engineering of dalvik applications are find out, whether the application contains malicious code, like ad/spyware, or some phone home functionality that sends data via a hidden channel to the vendor. Additionally one could query whether an application or the libraries it statically imports (in its APK container) has unpatched security holes, which means that the dex file was generated from vulnerable java code. A third reverse engineering perspective would check whether the code contains copied parts, which may violate GPL or other license agreements.

### WORKFLOW

Dalvik programmers follow a reoccurring workflow when coding their applications. In the default setup this involves javac, dx. There is no way back to java code once we compiled the code (see *Figure 2*). This differs from the java development model, where a decompiler is in the toolbox of every programmers. Our tool UNDX fills this gap, as shown in see *Figure 3*.



DESIGN CHOICES

Undx main task is to parse dex file structures. So before coding the tool there was a set of major design questions to be decided. The first was about the reuse grade of the parsing strategy, the second one was the library choice for generating java bytecode.

PARSING DEX FILES

DESIGN

The dexdump tool of the android SDK can perform a complete dump of dex files, it is used by UNDX, Table 3 lists the parameters that influenced the design of the parser. The decision was to use as much of useable information from dexdump, for the rest we parse the dex file directly. Figure 4 shows useful dexdump output, which is relatively easy to parse, compared to native Dex structures. On the other hand there are frequent omissions in the output of dexdump, such as the dump of array data (as in Figure 5).

Table 3: Parsing strategy

Table with 3 columns: dexdump, parsing directly, and a comparison of Speed, Control, and Available info.

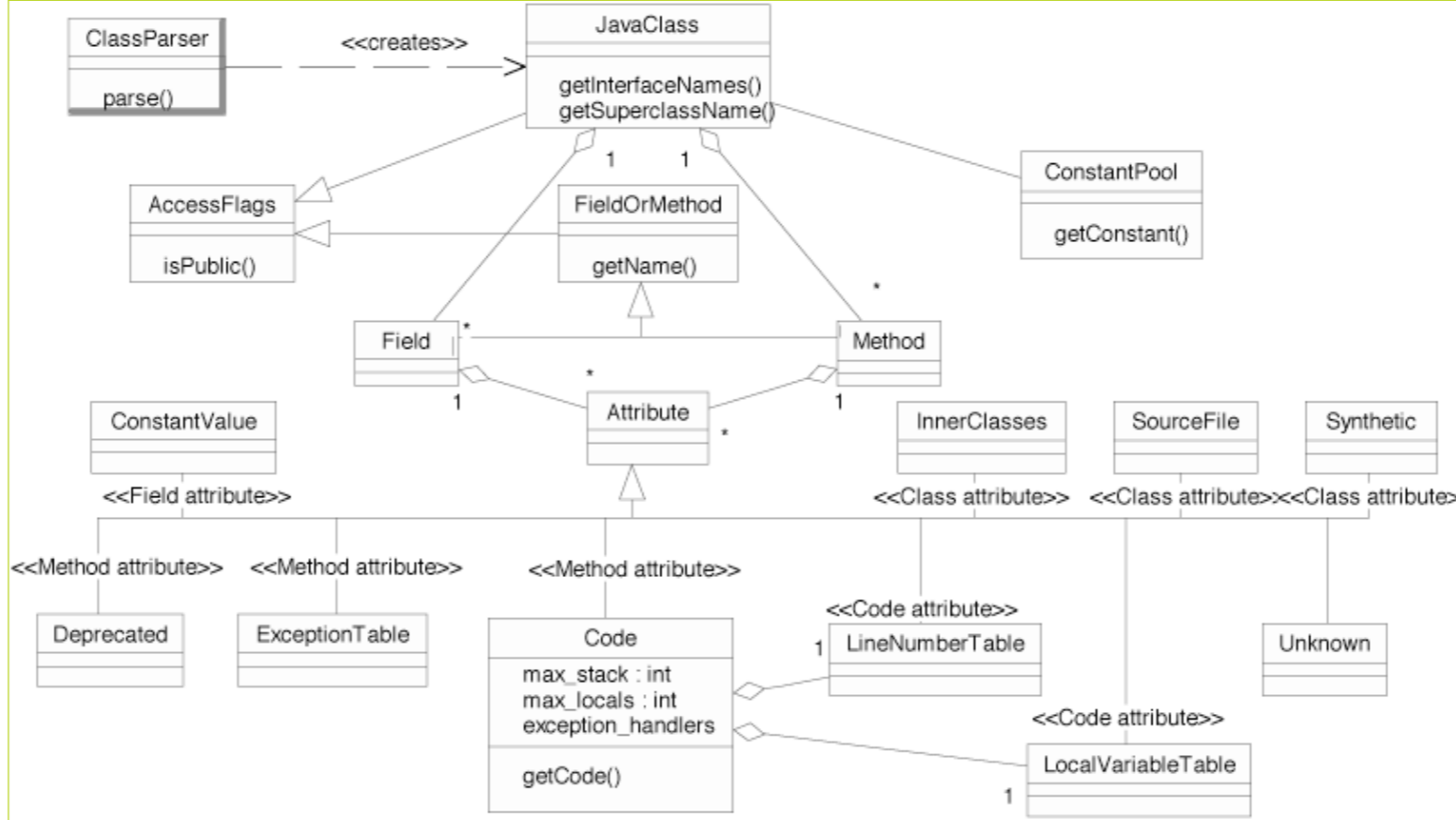
Figure 4: Dexdump output

```
#1 : (In LUndxTest;)
name : "main"
type : "[Ljava/lang/String;V"
access : 0x0009 (PUBLIC STATIC)
code : -
registers : 3
ins : 1
outs : 2
insns size : 20 16-bit code units
00024c: [[0024c] UndxTest.main([Ljava/lang/String;V
00025c: 2200 0200 1000: new-instance v0, LObj; // class#0002
000260: 7010 0000 0000 1000: invoke-direct {v0}, LObj;.<init>:()V // method#0000
000266: 1251 1000: const/4 v1, #int 5 // #5
000268: 6e20 0200 1000 1000: invoke-virtual {v0, v1}, LObj;.dot:(I)V // method#0002
00026e: 1301 ff00 1000: const/16 v1, #int 255 // #ff
000272: 6e20 0200 1000 1000: invoke-virtual {v0, v1}, LObj;.dot:(I)V // method#0002
000278: 1301 f000 1000: const/16 v1, #int 240 // #f0
00027c: 6e20 0200 1000 1000: invoke-virtual {v0, v1}, LObj;.dot:(I)V // method#0002
000282: 0a00 1000: return-void
```

Figure 5: Dexdump array dump output

```
name : "cclinit"
type : "[C"
access : 0x10008 (STATIC CONSTRUCTOR)
code : -
registers : 1
ins : 0
outs : 0
insns size : 34 16-bit code units
000318: [[000318] HD5.cclinit:()V
000320: 1200 1000: const/4 v0, #int 0 // #0
000322: 6900 0200 1000 1000: sput-object v0, LMD5;md5:LMD5; // field#0002
000326: 1300 1000 1000: const/16 v0, #int 16 // #10
00032a: 2300 0f00 1000: new-array v0, v0, [C // class#000f
00032e: 2600 0700 0000 1000: fill-array-data v0, @0000000e // @0000000e
000334: 6900 0000 1000: sput-object v0, LMD5;hexChars:[C // field#0000
000338: 0a00 1000: return-void
00033a: 0000 1000: nop // spacer
00033c: 0003 0200 1000 0000 3000 3100 3200 ... 1000: array-data (20 units)
catches : (none)
positions :
@0000 line-7
@0003 line-8
```

Figure 6: BCEL hierarchy



We chose the BCEL (<http://jakarta.apache.org/bcel/>) as bytecode backend, as it has a very broad functionality (compared to the potential alternatives like ASM and javassist), however this preference is solely based on the authors subjective view and experience with BCEL. Figure 6, which was taken from the BCEL documentation, shows the object hierarchy provided by the BCEL classes.

PROCESSING STEPS

Figure 7 shows the steps that are necessary to parse an APK back into a java bytecode representation. First global

Figure 7: Processing steps

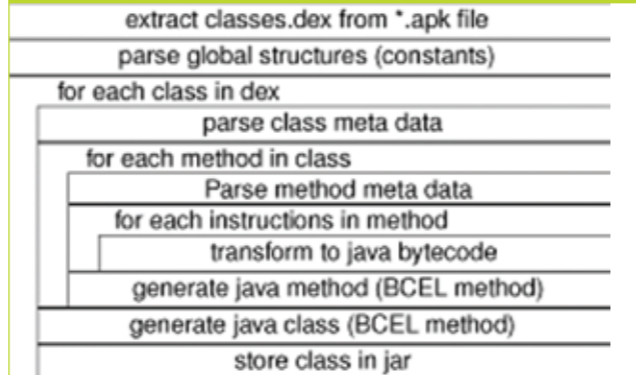


Figure 8: Acquire method meta data

```
private MethodGen getMethodMeta (ArrayList<String> al,
ConstantPoolGen pg,
String classname) {
for (String line : al) {
KeyValue kv = new KeyValue(line.trim());
String key = kv.getKey(); String value = kv.getValue();
if (key.equals(str_TYPE)) type = value.replaceAll("'", "");
if (key.equals("name")) name = value.replaceAll("'", "");
if (key.equals("access")) access = value.split(" ")[0].
substring(2);
allfound = (type.length() * name.length() * access.
length() != 0);
if (allfound) break;
}
Matcher m = methodtypes.matcher(type);
boolean n = m.find();
Type[] rt = Type.getArgumentTypes (type);
Type t = Type.getReturnType (type);
int access2 = Integer.parseInt(access, 16);
MethodGen fg = new MethodGen(access2, t, rt, null,
name, classname,
new InstructionList(), pg);
return fg;
}
```

Figure 9: Transforming the new-array opcode

```
private static void handle_new_array (String[] ops,
InstructionList il,
ConstantPoolGen cpg, LocalVarContext lvg) {
String vx = ops[1].replaceAll("'", "");
String size = ops[2].replaceAll("'", "");
String type = ops[3].replaceAll("'", "");
il.append(new ILOAD((short) lvg.didx2jvmdxstr(size));
if (type.substring(1).startsWith("L")
|| type.substring(1).startsWith("[") {
il.append(new ANEWARRAY(Utils.doAddClass(cpg, type.
substring(1))));
} else {
il.append(new NEWARRAY((BasicType) Type.getType(type.
substring(1))));
}
il.append(new ASTORE(lvg.didx2jvmdxstr(vx));
}
```

Figure 10: Transforming virtual method calls

```
private static void handle_invoke_virtual (String[] regs,
String[] ops,
InstructionList il, ConstantPoolGen cpg,
LocalVarContext lvg,
OpcodeSequence oc, DalvikCodeLine dcl) {
String classandmethod = ops[2].replaceAll("'", "");
String params = getparams(regs);
String a[] = extractClassAndMethod(classandmethod);
int metref = cpg.addMethodref(Utils.toJavaName(a[0]),
a[1], a[2]);
genParameterByRegs(il, lvg, regs, a, cpg, metref,
true);
il.append(new INVOKEVIRTUAL(metref));
DalvikCodeLine nextInstr = dcl.getNext();
if (!nextInstr.opname.startsWith("move-result")
&& !classandmethod.endsWith(")V")) {
if (classandmethod.endsWith(")J") ||
classandmethod.endsWith(")D") {
il.append(new POP2());
} else {
il.append(new POP());
}
}
}
```

APK structures are read, then the methods are processed. In the end the derived data is written to a jar file.

Processing of global structures: Processing the global structures involves extracting the classes.dex file from the APK archive (which is a zip container), and parsing global structures, like preparing constants for later lookup. In detail this step transforms APK meta information into relevant BCEL structures, for example retrieve the Dalvi String table and store its values in a JAVA constant pool.

Process classes: Transforming the classes involves splitting the combined meta data of the classes within a dex file into individual class files. For this purpose we parse the meta data, process the methods, by inspecting the bytecode and generate BCEL classes, as we now have all necessary meta data available and all methods of a class are parsed. The BCEL class object is then ready to be dumped into a class file, as entry of the output jar file.

Processing class Meta Data: This step includes extracting the meta data first, then transferring the visibility, class/interface, classname, subclass information into BCEL fields. The static and instance fields of each class have to be created, too.

Figure 11: Transforming sparse switches

```
String reg = ops[1].replaceAll(" ", "");
String reg2 = ops[2].replaceAll(" ", "");
DalvikCodeLine dclx = bll.getByLogicalOffset(reg2);
int phys = dclx.getMemPos();
int curpos = dclx.getPos();
int magic = getAPA().getShort(phys);
if (magic != 0x0200) { Utils.stopAndDump("wrong magic");
}
int size = getAPA().getShort(phys + 2);
int[] jumpcases = new int[size];
int[] offsets = new int[size];
InstructionHandle[] ihh = new InstructionHandle[size];
for (int k = 0; k < size; k++) {
jumpcases[k] = getAPA().getShort(phys + 4 + 4 * k);
offsets[k] = getAPA().getShort(phys + 4 + 4 * (size + k));
int newoffset = offsets[k] + curpos;
String zzzz = Utils.getFourCharHexString(newoffset);
ihh[k] = ic.get(zzzz);
}
int defaultpos = dclx.getNext().getPos();
String zzzz = Utils.getFourCharHexString(defaultpos);
InstructionHandle theDefault = ic.get(zzzz);
il.append(new ILOAD(locals.didx2jvmdxstr(reg)));
LOOKUPSWITCH ih = new LOOKUPSWITCH(jumpcases, ihh, theDefault);
il.append(ih);
```

Figure 15: Decompilation

```
public class WebDialog extends Dialog
{
public WebDialog(Context arg0)
{
super(arg0);
Object obj = JVM INSTR new #14 <Class WebView>;
((WebView) (obj)).WebView(arg0);
webView = ((WebView) (obj));
obj = webView;
obj = ((WebView) (obj)).getSettings();
boolean flag = true;
((WebSettings) (obj)).setJavaScriptEnabled(flag);
obj = webView;
setContentView(((android.view.View) (obj)));
obj = "Welcome";
setTitle(((CharSequence) (obj)));
}

public void loadUrl(String arg0)
{
WebView webview = webView;
webview.loadUrl(arg0);
}
}
```

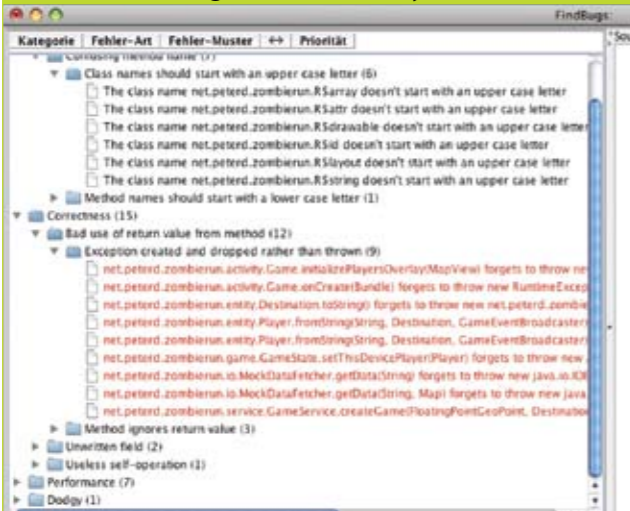
Figure 12: Dalvik Code

```
type : 'CJMS;'
access : 0x0009 (PUBLIC STATIC)
code :
registers : 1
lins : 1
uops : 1
insns size : 14 16-bit code units
DexOpCIP:
0000:00: 6200 0200 0000: sget-object v0, LMS;:ms:LMS; // Field#0002
0001:01: 7300 0000 0000: if-nez v0, :loop // #0003
0002:02: 2200 0200 0000: new-instance v0, LMS; // class#0002
0003:03: 7030 0200 0000: invoke-direct (v0, LMS;:<init>:()V // method#0000
0004:04: 6900 0200 0000: sput-object v0, LMS;:ms:LMS; // Field#0002
0005:05: 6200 0200 0000: sget-object v0, LMS;:ms:LMS; // Field#0002
0006:06: 1100 0000 0000: return-object v0
catches : (none)
position :
00000 line-24
00000 line-26
```

Figure 13: JVM Code

```
public static MS getInstance();
Code:
0: getstatic #14; //Field ms:LMS;
3: astore_0
4: aload_0
5: ifnonnull :20
6: new #4; //class MS
11: astore_0
12: aload_0
13: invokespecial #73; //Method <init>:()V
16: aload_0
17: putstatic #14; //Field ms:LMS;
20: getstatic #14; //Field ms:LMS;
23: astore_0
24: aload_0
25: return
```

Figure 14: Static Analysis

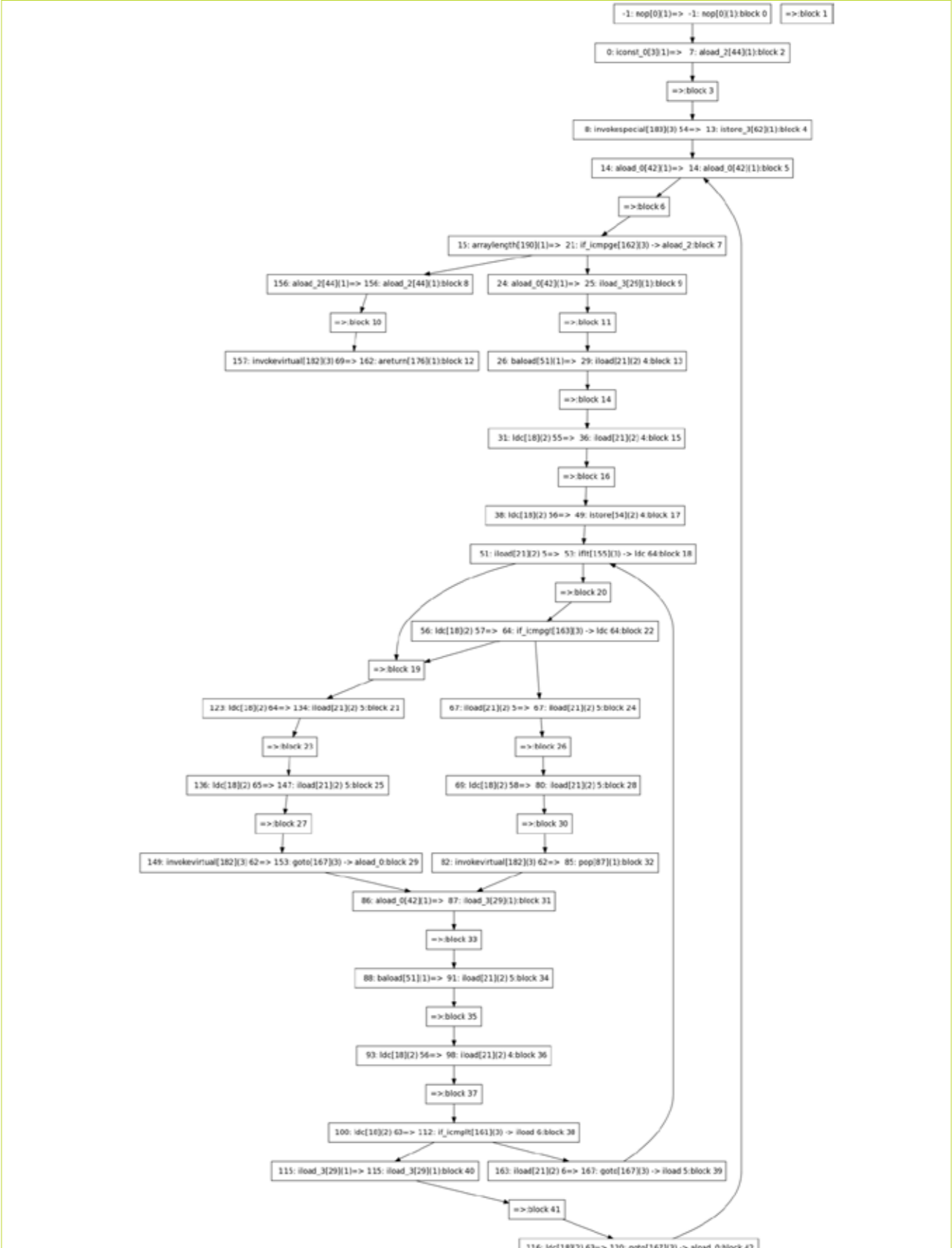


Process the individual methods: The major work of UNDX is performed in transferring the Dalvik bytecode back into JVM equivalents. So first we extract the method meta data, then parse all the Instructions and generate BCEL methods for each Dalvik method. This includes transforming method meta data to BCEL method structures, extracting method signatures setting up local variable tables, and mapping Dalvik registers to JVM stack positions. A source snippet for this is shown in Figure 8.

Generating the java bytecode instructions: The details for creating BCEL instructions from Dalvik instructions are very work-intensive. First BCEL InstructionLists are created, then NOP proxies for every Dalvik instruction to handle forward jump targets are prepared. Then for every Dalvik instruction add an equivalent JVM bytecode block to the JVM InstructionList. In this conversion loop UNDX spends most of its time. Not every instruction can be processed one-to-one, as some storage semantics are differing between Dalvik and JVM, as shown in Figure 9, Figure 10 and Figure 11. The instructions shown in Figure 12 and Figure 13 illustrates the transformation results. To achieve this result we have to comply to some invariant constraints, we have to assign sound Dalvik regs to jvm stack positions.

To violate the JVM verifier as less as possible we want to obey stack balance rule, when processing the opcodes. Very important also is to provide proper type inference of the object references on the stack (reconstruct flow of data assignment opcodes). This is often tricky and fails in the set of cases, where the Dalvik reused registers for objects of differing types. This detail illustrates well how hardware and memory constraints in mobile devices influenced the design of the Dalvik architecture.

Figure 16: Graph With DIA



Store generated data in BCEL structures: After all methods in all classes are parsed, processing is finished, and as result we have a class file for each defined class in the dex file.

**Static analysis of the code**

Now that we have bytecode generated from the Dalvik code, what can we do with it. We could analyze the code with static checking tools, like (findbugs) to find programming bugs, vulnerabilities, license violations with tool support (see Figure 14). If we are an experienced reverse engineer and already learned that fully automated tools are not the ultimate choice in RE, we stuff the class files in a decompiler (JAD, JD-GUI), see Figure 15 to receive JAVA-like code to speed up program understanding, which is the reverse engineers primary goal. Be aware, that you receive structural equivalent and not a 100 percent verbatim copy of the original source, as some differences due to heavy transformation processes inbetween show their effect, such as reuse of stack variables.

In certain cases it is recommended to use class file disassembler (javap), when the decompiler was not able to complete due to heavy use of obfuscation.

Although real reverse engineers prefer code, UNDX can also compete in the RE softball league, using more graphs and consume less brain. If you want that instead, write a 20 liner groovy script, and transfer the nodes and arrows

of the control flow graph (like the one offered by findbugs) into a nice graph in the graphing language of your choice. Figure 16 shows that approach using DIA.

**SUMMARY AND TRIVIA**

UNDX consists of about 4000 lines of code, which are written in JAVA, only external dependency is BCEL. It uses the command line only, but you could write a GUI and contribute it to the project, as the licensing is committer-friendly GPL. The code is available at <http://www.illegalaccess.org/undx/>.

At this point we thank *Dan Bornstein* (again), for suggesting the UNDX name. •

**ABOUT THE AUTHOR**

Marc Schönefeld is a known speaker at international security conferences since 2002. His talks on Java-Security were presented at Blackhat, RSA, DIMVA, PacSec, CanSecWest, HackInTheBox and other major conferences. In 2010 he hopefully finishes his PhD at the University of Bamberg. In the daytime he works on the topic of Java and JEE security for Red Hat. He can be reached at [marc@illegalaccess.org](mailto:marc@illegalaccess.org).



- CONFERENCES, SEMINARS, TRAININGS
- STANDARDS, FRAMEWORKS & BEST PRACTICES
- PROFESSIONAL CERTIFICATIONS
- NETWORKING & SOCIAL EVENTS
- VOLUNTARY & EDUCATIONAL OPPORTUNITIES

**MEMBERSHIP BENEFITS**

- ENHANCE YOUR KNOWLEDGE
- ADVANCE YOUR CAREER
- STAND OUT WITH ISACA CERTIFICATIONS
- CONNECT AND NETWORK



**ISACA membership**—more than 95,000 strong worldwide—is characterized by its diversity. Members live and work in more than 160 countries and cover a variety of professional IT-related positions. Some are in the most senior ranks, others are at middle management levels and still others are new to the field. They work in nearly all industry categories, including financial and banking, public accounting, government and the public sector, utilities and manufacturing.

**Upcoming Training and Events**

**25th Anniversary Celebration and Awards Dinner - 26 October 2010**

**COBIT 5 and IT Governance 9 November 2010**

**Information Security Workshop 11—12 November 2010**

**IT Audit & Assurance, IT Security, IT Risk Management and IT Governance Training Week 29, 30 Nov and 1 Dec 2010**

**Advance IT Audit and Assurance Workshop 2—3 December 2010**

**Get Certified**

**Certified Information Systems Auditor (CISA)**



The CISA certification is known worldwide as the recognized achievement for those who control, monitor and assess an organization's information technology and business systems.

**Certified Information Security Manager (CISM)**



The management-focused CISM is a unique certification for individuals who design, build and manage enterprise information security programs. CISM is the leading credential for information security managers.

**Certified in the Governance of Enterprise IT (CGEIT)**



CGEIT recognizes a wide range of professionals for their knowledge and application of enterprise IT governance principles and practices.

**Certified in Risk and Information Systems Control (CRISC)**



**CRISC** (pronounced "see-risk") is designed for IT professionals who have hands-on experience with risk identification, assessment and evaluation; risk response; risk monitoring; IS control design and implementation; and IS control monitoring and maintenance.



# UBUNTU for Non-Geeks

## A Pain-Free, Get-Things-Done Guide



by Rickford Grant with Phil Bull

Review by Dhillon Andrew Kannabhiran

With a title like 'Ubuntu for Non-Geeks' the target audience for this book is clearly not the seasoned HITB Magazine reader. That being said, with the holiday season just around the corner, this book would certainly be a great gift for someone looking to get his or her feet wet in the world of the penguin.

Broken down into 21 easy to follow chapters, the book kicks off as most other introductory titles do, with a brief intro to Linux in general, providing the reader with the usual 'about Linux' sections followed by some background information on the Ubuntu distribution in particular. Like its predecessors, the book is bundled with an Ubuntu live CD and this 4th edition ships with Ubuntu 10.04 (Lucid Lynx).

Unlike other 'Linux for beginners' type books however, Ubuntu for Non-Geeks is written to be used as both a reference guide or to be read cover-to-cover. It assumes the reader is already somewhat familiar with computers in general and certainly seasoned in Microsoft Windows.

The book is designed to teach by taking the reader through various 'projects'. Presented in a tutorial style, these follow-along guides are designed to get the reader involved in solving a specific task in order to learn and more importantly understand how things in Linux work.

Projects start off with the very basics – customizing your desktop's look and feel (thus getting exposed to GNOME desktop's panels and widgets) to getting connected and online. This is then followed by slightly more advanced projects – things like keeping system and application software up to date via the Ubuntu Software Center, a chapter on the Terminal and introductory commands and projects dealing with things like burning DVDs and getting your iPod or iPhone to work with your Linux system. There's even a chapter on getting anti virus software installed, configuring a basic software firewall (Firestarter) and getting encrypted files and folders set up.

There was also a chapter on Linux gaming, although I'm not sure how many Linux adopters are coming over for the games. Real gamers would probably opt for a dual boot set up anyway although for the casual gamer, the projects on getting Wine installed or running Windows within a virtual machine would probably be of interest.

While this book claims to be aimed at the non-geek, as mentioned earlier it does assume that the reader is already familiar with computers in general and that they would understand certain specific IT terms. That being said, the slightly more technically inclined, who have always wanted to try out Linux but didn't want to find themselves 'stuck' trying to get something to work, would definitely find this book useful with its step by step project based approach which makes learning Linux a whole lot easier.

**“Presented in a tutorial style, these follow-along guides are designed to get the reader involved in solving a specific task in order to learn and more importantly understand how things in Linux work.”**

**Ubuntu for Non-Geeks**  
4th Edition

Author: Rickford Grant & Phil Bull

Publisher: No Starch Press

Pages: 496, w/CD

ISBN: 978-1-59327-257-9



“When an individual starts following the constructive path, the journey becomes interesting and success inevitably follows.”

**ADITYA K. SOOD**  
Founder *SecNiche Security*

Zarul Shahrin talks to **ADITYA K. SOOD**, active speaker for RSA, XCON, writer for Hakin9, HITB and the founder of SecNiche Security and a PhD candidate at Michigan State University.

**H** **Hi Aditya, how are you?**  
Hi Zarul, I am fine and going good.

**Maybe you can share with our readers something about yourself and how did you get involved with computer security.**

I started working in the computer security from my college days, even though the journey has not been easy. As we know, success comes at its own costs. However, in time and eventually burning midnight oil enables you to learn a lot of things. I spent a lot of time understanding the crux of the security field and kept on motivating myself during those unpleasant times which is unavoidable in every field. I started learning a lot of things in a practical manner by perusing and studying the research of other brilliant researchers and people in the security community. Perseverance and “Never Give up” attitude helped me to acquire the basic knowledge that I could use as a launch pad. I sincerely believe in serving the security community as we all learn a lot from it. So, I feel I have the responsibility to give back to the community by engaging in productive security researches.

**Before pursuing your PhD at Michigan, what did you work as?**

Well, I worked for COSEINC which is a vulnerability research and security consulting company and was primarily engaged in vulnerability research area. Many countries have taken initiatives to address the risks of potential vulnerabilities persisting in their running systems.

**Which area of security interest you the most?**

I have keen interest in the diverse facets of computer security. The concern for security instills a sense of responsibility in me. My work is focused on web security research, malware analysis, and vulnerability research. In addition, I like to do security testing which includes web application security assessments, penetration testing, and source code reviews. Testing itself helps in detecting vulnerabilities across a wide range of devices and vendor products. I try to contribute to the community by publishing papers and articles on my website, magazines and journals. I believe in sharing my knowledge and thoughts because it helps me to set a platform of communication between two parties to enhance the learning experience.

**Few years back, you have been the victim of what I would call as mailing list “Troll”. How did you take it personally?**

Yes Zarul, definitely. I am very open to this. I did not let myself get distracted by unfair criticism. Personally, I believe that human efforts should be constructive in nature. So, I decided not to waste my time by indulging in the rogue communication that was happening in the mailing lists. I firmly believe that criticism should not deter any individual from pursuing his career path. The important thing is to remain committed to your goals. I think it happens to many genuine professionals. We have online democracy where everyone enjoys freedom of speech. Freedom also endows a responsibility to the individuals to adhere to the protocols of communication. Indulging in feckless criticism does not lead an individual anywhere. During that course of time, I concentrated on work with the best of my abilities without getting distracted into communication with the “Troll”. When an individual starts following the constructive path, the journey becomes interesting and success inevitably follows.

**Do you personally know who this person was and what did he has against you?**

At this point of time, it does not matter because I have left those things far behind. Yes, I knew the person very well but I do not believe in unfolding the history which is buried a long time ago. The question is, “Is it necessary to intermingle your personal inflexibility with professional couture?” If somebody does that, it will be hard to determine the authenticity of that person’s personality. It is always possible to convey disagreement with one’s views

I sincerely believe in serving the security community as we all learn a lot from it. So, I feel I have the responsibility to pay back to the community by engaging in productive security research.

more artistically and in a good manner. Healthy criticism is the pre-cursor to new knowledge. In general, it is human fallacy and it is hard to conquer it.

**So I guess you consider this as nothing more than a distraction?**

I do not let myself get restricted by these minute distractions. The real point lies in Walking Tall with efforts directed towards constructive approach and focused on learning new things. You become more mature with the passage of time and God showers HIS blessings if you hold the element of purity and truth. This is my definition of professionalism.

**How about your decision to leave your job at COSEINC and pursue your PHD at Michigan. Is this something that you have planned earlier?**

Its not about leaving the job. Actually, I believe that there is a gap between academia and industry. I am just putting my efforts to fill that gap as much as I can so that we can come up to a single entity and collaborative research. This helps us in establishing a bridge and simulation of ideas between two different worlds. For your information, I am still working in the industry-specific research.

**So, are you planning to stay in Academia after your PhD?**

It is a good learning experience to understand the artifacts of academia. As I mentioned earlier, that bridging a gap is my main target. I believe both aspects of learning is important.

**What is the focus of your research at Michigan?**

My research is based on solving practical problems rather than theoretical in the field of web security and malware. At present, I am concentrating towards web malware analysis and impacts on real time environment. Web malware is a severe problem and we require more research and analysis at core level rather than pointing out the generic nature. I am driven towards this kind of work. I think potential and coherent research is required to get inline with web malware issues.

**Please elaborate more about this.**

Web malware is a sophisticated piece of malicious code that is injected in websites by exploiting vulnerabilities to execute "drive by downloads" attacks to infect machines. Web malware has different facets but its sole aim is to wreak damage by stealing sensitive information. The attack vector of latest malware attacks can be categorized into three broad categories

1. Infection through Third Party Content Inclusion (Malvertisements, Obfuscated Links etc)
2. Mass Outbreaks by Datacenter Compromises (Mass Infection - Can be SQL, XSS etc)
3. Exploitation of trust in Social Networks

Malware is exploiting the trends of increasing third party content inclusion from various resources on the Internet. Primarily, a well activated website renders content from different websites and uses that content as a centralized point for information sharing. However, most of the feeds and the content are not scrutinized prior to inclusion on the primary website which exposes them to malware. Vulnerabilities play a critical role in the dissemination of malware. Lastly, data center infection also results in mass compromise of websites. Datacenters are primarily controlled by botnets. My continuous analysis reveals the fact that admin scripts are exploited at a large scale solely for infecting servers. Generating rogue profile in social networks to spread malware is the biggest ongoing infection attack vector to exploit the trust of social connection. Furthermore, URL shorteners used in Twitter application also enable malware writers to hide the actual URL content and compress it.

**With 4 years working experience as a security analyst under your belt for different companies based in Asia, what do you have to say about this region when it comes to computer security?**

First of all, I sincerely believe that computer security is a global concern. Internet has facilitated strong inter-linkages among the various operating entities. There is a pressing need to secure the inter linkages from the fraudulent activities undertaken by the hackers. Their harmful actions have the potential to inflict permanent damages. We do have the required regulatory framework in place to thwart the actions of the hackers. But the enforcement of cyber laws is not stringent enough to combat the attacks in the real world. Thus, we have witnessed a significant rise in web malware related activities. Asia has been at the forefront of exploitation. Governments are also taking aggressive steps in Asian countries to build cyber armies. It can be considered as a proactive defense but the real solution still remains elusive. The computer security issues are not country specific but are a global problem. All the countries have to join hands in order to design standard benchmarks for fighting against the evil and perils of cyber crimes.

**We are seeing a growing number of hacking websites promoting illegal activities in this region compare to a few years ago. Do you think the local government agencies should start taking down these websites and the people involve?**

Your question absolutely hits the nail. We have noticed a huge increase in illegal activities. It makes me believe that machinations of hackers are complex. We do have a vague idea about their modes of operations but it is increasingly becoming difficult to comprehend the adverse impacts of their actions. There has been a tremendous change in the methodologies adopted by the hackers to attack the websites and design malicious codes. The repercussions of their actions can be detrimental for organizations, firms and countries worldwide. Asia is most exposed to the threat of defacement of websites. Asian countries have become prime point for spreading malware followed by Russia. Chinese and Russian malwares are the most destructive ones. The lack of convergence in cyber laws among different countries is a primary obstacle and concern which hinders their effectiveness in tackling cyber crimes. Local governments should act rigorously in order to combat these cyber crimes. The increased dependence on computers has created an urgent need for robust security mechanisms.

**How about the high number of fraud cases originating from this region?**

Yes that is true. Asia is one of the most targeted markets for frauds when it comes to cyber crimes. Countries like China, Taiwan, and Korea are key players in Asia for effectively executing the fraudulent activities on the World Wide Web. Attackers have become adept at using sophisticated methods to conduct online attacks for stealing information. However, most of the botnet attacks are driven towards financial institutions' websites. These types of attack scenarios are termed as dedicated attacks where the destination are pre-selected.

The Asian hackers are tracking the rabbit hole with efficient structural and technological components which helps them to derive strong methodology of hacking. In the past, I have presented at leading security and hacking conferences in China such as XCON and Excalibur. To the best of my knowledge, Chinese hackers are increasingly becoming successful in deploying robust and mature attacks to inflict serious damages. But the modes of their operation still remain hidden and as a result the free flow of information is restricted.

**Thank you Aditya**

You're welcome.

**ADITYA K. SOOD** is a PhD candidate at Michigan State University. He has already worked in the security domain for Armorize, COSEINC and KPMG. He is a founder of SecNiche Security, an independent security research arena. He has been an active speaker at conferences like RSA (US 2010), TRISC, EuSecwest, XCON, Troopers, OWASP AppSec, FOSS, CERT-IN etc. He has written content for HITB Ezine, Hakin9, Usenix Login, Elsevier Journals, Debugged! MZ/PE.

**The Asian hackers are tracking the rabbit hole with efficient structural and technological components which helps them to derive strong methodology of hacking.**



## 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](mailto:media@hackinthebox.org)