International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

Exploit Development Research on x86 Windows Application: Buffer Overflow

Geogen George[1], Sivasundaram R[2]

[1]Cyber Security Research Center, SRM University, Chennai, India

[2]M.Tech, Information Security and Cyber Forensics, SRM University, Chennai, India

Abstract: ***The Best way to find vulnerability present with an application is to presume you as Hacker and act accordingly. Hackers always find one or other way to bypass all the security measures taken by the developers. So before releasing a developed software application, it's always advised to do Vulnerability Testing. In this paper, I'm briefly discussing Advanced Testing Methodology by how hacker may bypass all exploit mitigation techniques and developing an exploit for it, as the proof of concept code.***

Keywords: Vulnerability Testing; Exploit Development; Bypass DEP, ASLR, SEH, Egg Hunting.

1. Introduction

Buffer overflows are vulnerabilities. They continue to be a problem for software security. Proper securing of software should seek to ensure goals of confidentiality, integrity, authentication, availability, and non- repudiation. Buffer overflow vulnerabilities affect the assurance of several of these goals. Buffer overflows are not always easy to discover and even when an overflow is discovered, it can be difficult to reverse its effects. In various locations and settings, buffer overflow attacks have been launched and have caused problems. Buffer overflow attacks are mostly targeted towards popular sites and software. From Microsoft software to social networking sites, there have been several attacks which have been found to be caused by a buffer overflow exploit.

Therefore, Finding the Vulnerability and patching them before releasing, is more important. Though Organization follows software testing for bugs, there is no much attention paid towards vulnerability finding. A Vulnerable application may cause the attacker to exploit it in many ways. Exploit may be Denial of Service, Remote Shell, and User Privilege Escalation, which gives complete control to Hackers and ends in Security breach.

Objective of this project work is to design Testing Mechanism and Application for to find vulnerability and develop an exploit against Windows x86 Application. Microsoft has its own Exploit Mitigations techniques which help to prevent buffer overflowing in memory. Ultimately, I'll be explaining how hacker may bypass all these mitigation, finding the Vulnerability of an Application, with Proof-of-Concept (POC) Exploit. So, it helps developers to release their software product, buffer overflow vulnerability-free.

2. Literature Review

A Research has stated in multiple circumstances that software and application-layer vulnerabilities, intrusions, and intrusion attempts are on the rise. Software-based vulnerabilities, especially those that occur over the Web are extremely difficult to identify and detect.

"Today, over 70 percent of attacks against a company's network come at the Application layer not the Network or System layer".—The Gartner Group

A. Buffer Overflow

A buffer overflow is vulnerability, a weakness which may allow a threat to exploit the software program. A simple analogy that may describe what a buffer overflow is may be overfilling a glass with water. In this case, the glass is compared to a buffer and the water is compared to the various values that may be put into a buffer. If there is too much water put into the glass, the water in turn overflows onto the surface holding the glass causing a mess. In this analogy, the surface holding the glass can be compared to a computer's memory space. When the contents of a buffer are overflowed, the overflow can overwrite a portion of a computer's memory. The information stored at this memory location could possibly be lost forever. Included in this information that is lost is the list of instructions that tell the program, which has placed information in the buffer, where to go and what to do next. The program will not be able to pick up where it left off or finish its tasks as it is lost.[1]

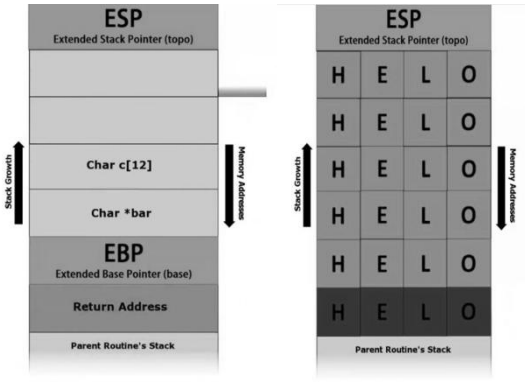

Figure 1: Buffer Overflow Illustration

B. Data Execution Prevention

Data Execution Prevention (DEP) is a set of hardware and software technologies that perform additional checks on

memory to help prevent malicious code from running on a system. The primary benefit of DEP is to help prevent code execution from data pages. Typically, code is not executed from the default heap and the stack. Hardware-enforced DEP detects code that is running from these locations and raises an exception when execution occurs. Software-enforced DEP can help prevent malicious code from taking advantage of exception-handling mechanisms in Windows.

Hardware-enforced DEP relies on processor hardware to mark memory with an attribute that indicates that code should not be executed from that memory. DEP functions on a per-virtual memory page basis, and DEP typically changes a bit in the page table entry (PTE) to mark the memory page.

Software-enforced DEP runs on any processor that can run Windows XP SP2. By default, software-enforced DEP helps protect only limited system binaries, regardless of the hardware-enforced DEP capabilities of the processor.[3]

C. Address Space Layout Randomization

Address space layout randomization (ASLR) is a computer security technique involved in protection from buffer overflow attacks. In order to prevent an attacker from reliably jumping to a particular exploited function in memory, ASLR involves randomly arranging the positions of key data areas of a program, including the base of the executable and the positions of the stack, heap, and libraries, in a process's address space.

Microsoft's Windows have ASLR enabled for only those executable and dynamic link libraries specifically linked to be ASLR-enabled. For compatibility, it is not enabled by default for other applications. The locations of the heap, stack, Process Environment Block, and Thread Environment Block are also randomized. A security whitepaper from Symantec noted that ASLR in 32-bit Windows may not be as robust as expected, and Microsoft has acknowledged a weakness in its implementation.[4]

D. Structured Exception Handling Overwrite Protection

An exception is an event that occurs during the execution of a program, and requires the execution of code outside the normal flow of control. There are two kinds of exceptions: hardware exceptions and software exceptions. Hardware exceptions are initiated by the CPU. They can result from the execution of certain instruction sequences, such as division by zero or an attempt to access an invalid memory address. Software exceptions are initiated explicitly by applications or the operating system. For example, the system can detect when an invalid parameter value is specified.[5]

Structured exception handling is a mechanism for handling both hardware and software exceptions. Therefore, your code will handle hardware and software exceptions identically. Structured exception handling enables you to have complete control over the handling of exceptions, provides support for debuggers, and is usable across all programming languages and machines. Vectored exception handling is an extension to structured exception handling. For example, a termination handler can guarantee that clean-up tasks are performed even if an exception or some other error occurs while the guarded body of code is being executed.

SafeSEH is only a linker that can be used at the compilation process of a program/software in Windows system. When the SafeSEH is used, the application will generate a table that contain all memory address that will be used by itself and also save the addresses of the SEH on the modules used. This means, when an exploitation that utilize the POP POP RETN command happen, the address that used to bring the SEH to the POP POP RETN address won't work because the address is not recorded in the table generated by the SafeSEH and the exploitation will failed.[5]

3. Design

We Proposing a Testing Mechanism for Windows x86 Application, Considering how an attacker may discover any vulnerability present with the software. If so, Vulnerability Research engineer can exploit it with Proof-of-Concept code. This Mechanism includes how attacker may bypass all the Windows Exploit Mitigation techniques such as DEP, ASLR, SEHOP, SafeSEH. This will help an Organization to do self-uditing before releasing the Software for commercial/public use.

For developing an exploit, there are six modules that we need to follow. They are Fuzzing, Controlling EIP, Locating space for our shellcode, Identifying Bad characters, Redirecting the execution flow, Generating Payload using Metasploit Framework.

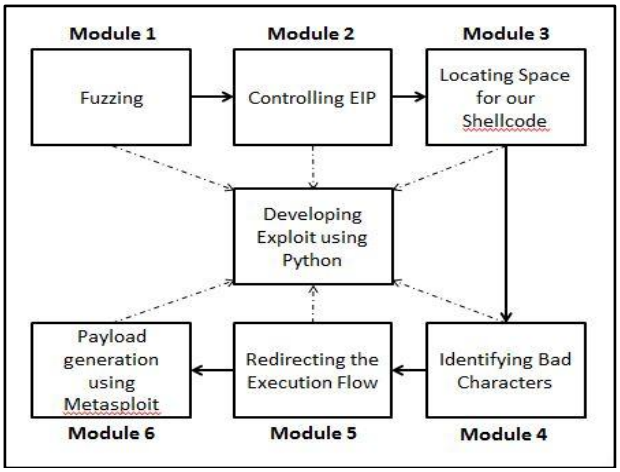

Figure 2: Solution Architecture for Stack based Buffer Overflow

Fuzzing involves sending malformed data into application input and watching for unexpected crashes. An unexpected crash indicates that the application might not filter certain input correctly. This could lead to discovering an exploitable vulnerability.

Controlling EIP register is a crucial step of exploit development. Generally, There are two techniques used for mapping EIP. One is Binary tree analysis and another is Sending a Unique string. In this paper, we are implementing

Sending Unique string. Using Pattern Create and Pattern Offset, We can map the EIP values.

Locating Space for our shellcode, A standard reverse shell payload requires about 350---- 400 bytes of space. We must check the availability of memory space after we map EIP register. Based on the payload we use, Memory space can be determined.

Bad Characters, Depending on the application, vulnerability type, and protocols in use, there may be certain characters that are considered "bad" and should not be used in your buffer, return address, or shellcode.

Redirecting the Flow of control, The value of ESP changes, from crash to crash. If we can find an accessible, reliable address in memory that contains an instruction such as JMP ESP, we could jump to it, and in turn end up at the address pointed to, by the ESP register, at the time of the jump.

Msfpayload, Metasploit Frameworks provides us with tools and utilities which make generating complex payload a simple task.

The msfpayload command can auto generate over 320 shell code payload options. We also need to provide the msfencode script to specify the bad characters we wish to avoid, in the resulting shell code.

While implementing each of the above modules, Applicaion tester crafts the python exploit shellcode for the final execution.
Along with this, Immunity Debugger plug-in named Mona.py aid us in several ways. This script will help us identify modules in memory that we can search for such a "return address", which in our case is a JMP ESP command. We will need to make sure to choose a module with the following criteria:
i)No memory protections such as DEP and ASLR present.
ii)Has a memory range that does not contain bad characters.

Syntax:!mona modules
A. Bypassing SEH
An exception handler is a piece of code that is written inside an application, with the purpose of dealing with the fact that the application throws an exception. Windows has a default SEH (Structured Exception Handler) which will catch exceptions. If Windows catches an exception, you'll see a "xxx has encountered a problem and needs to close" popup. This is often the result of the default handler kicking in. It is obvious that, in order to write stable software, one should try to use development language specific exception handlers, and only rely on the windows default SEH as a last resort. When using language EH's, the necessary links and calls to the exception handling code are generate in accordance with the underlying OS. (and when no exception handlers are used, or when the available exception handlers cannot process the exception, the Windows SEH will be used. (Unhandled Exception Filter)). So in the event an error or illegal instruction occurs, the application will get a chance to catch the exception and do something with it. If no exception handler is defined in the application, the OS takes over, catches the exception, shows the popup (asking you to Send Error Report to MS).

This structure (also called a SEH record) is 8 bytes and has 2 (4 byte) elements :

- A pointer to the next exception_registration structure (in essence, to the next SEH record, in case the current handler is unable the handle the exception)
- A pointer, the address of the actual code of the exception handler. (SE Handler)

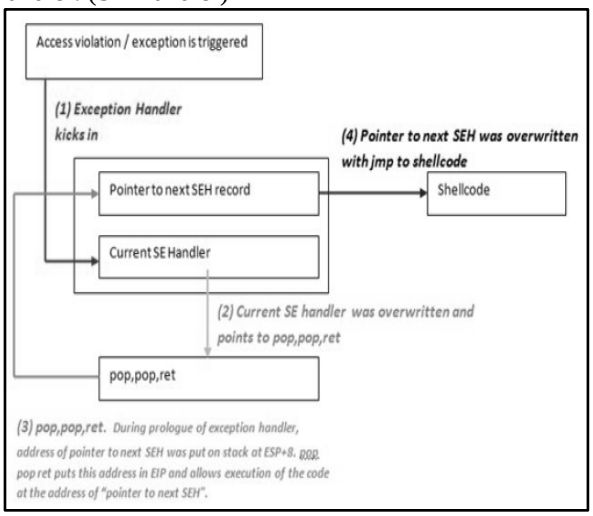

Figure 3: SEH bypassing

If we can overwrite the pointer to the SE handler that will be used to deal with a given exception, and we can cause the application to throw another exception (a forced exception), we should be able to get control by forcing the application to jump to your shellcode (instead of to the real exception handler function). The series of instructions that will trigger isPOP,POP,RET.[6]

B. Bypassing ASLR
Windows Vista, 2008 server, and Windows 7 offer yet another built-int security technique (not new, but new for the Windows OS), which randomizes the base addresses of executables, dll's, stack and heap in a process's address space (in fact, it will load the system images into 1 out of 256 random slots, it will randomize the stack for each thread, and it will randomize the heap as well). This technique is called ASLR (Address Space Layout Randomization).

The addresses change on each boot. ASLR is enabled by default for system images (excluding IE7), and for non-system images if they were linked with the /DYNAMICBASE link option (available in Visual Studio 2005 SP1 and up, and available in VS2008).

You can choose other system module which aren't with ASLR protection for developing an exploit. This can be done by using mona.py in immunity debugger. This plug-in will help us to identify which are all modules present with the system without ASLR protection.

C. Win32 Egg Hunting
Egg hunting is a technique that can be categorized as "staged shellcode", and it basically allows you to use a small amount

of custom shellcode to find your actual (bigger) shellcode (the "egg") by searching for the final shellcode in memory. In other words, first a small amount of code is executed, which then tries to find the real shellcode and executes it.
The decision to use a particular egg hunter is based on

- Available buffer size to run the egg hunter
- Whether a certain technique for searching through memory works on your machine or for a given exploit or not. You just need to test.

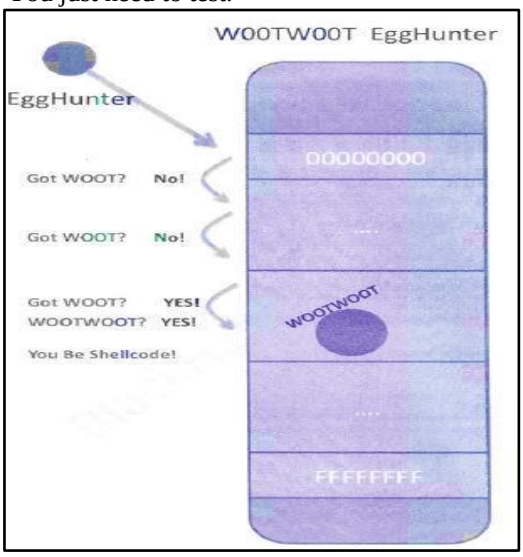

Figure 4: Egg Hunting

The tag used in this example is the string w00t. This 32 byte shellcode will search memory for "w00tw00t" and execute the code just behind it. This is the code that needs to be placed at esp.
When we write our shellcode in the payload, we need to prepend it with w00tw00t (= 2 times the tag – after all, just looking for a single instance of the egg would probably result in finding the second part of egg hunter itself, and not the shellcode).

First, locate jump esp. I'll use 0x7E47BCAF (jmpesp) from user32.dll (XP SP3).

Change the exploit script so the payload does this:

- Overwrite EIP after 710 bytes with jmpesp
- Put the egghunter at ESP. The egghunter will look for "w00tw00t"
- Add some padding (could be anything.nops, A's.)
- Prepend "w00tw00t" before the real shellcode.
- Write the real shellcode[6]

D. Bypassing DEP
Since we cannot execute our own code on the stack, the only thing we can do is execute existing instructions/call existing functions from loaded modules and use data on the stack as parameters to those functions/instructions.

These existing functions will provide us with the following options :

- Execute commands (WinExec for example)
- Mark the page (stack for example) that contains your shellcode as executable (if that is allowed by the active DEP policy) and jump to it
- Copy data into executable regions and jump to it. (We may have to allocate memory and mark the region as executable first)
- Change the DEP settings for the current process before running shellcode.

When we have to bypass DEP, we'll have to call a Windows API. The parameters to that API need to be in a register and/or on the stack. In order to put those parameters where they should be, we'll most likely have to write some custom code.

If one of the parameters to a given API function is for example the address of the shellcode, then you have to dynamically generate/calculate this address and put it in the right place on the stack. You cannot hardcode it, because that would be very unreliable

These are the most important functions that can help you to bypass/disable DEP

1) VirtualAlloc(MEM_COMMIT+PAGE_READWRITE_EXECUTE) + copy memory. This will allow you to create a new executable memory region, copy your shellcode to it, and execute it. This technique may require you to chain 2 API's into each other.
2) HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc() + copy memory. In essence, this function will provide a very similar technique as VirtualAlloc(), but may require 3 API's to be chained together.
3) SetProcessDEPPolicy(). This allows you to change the DEP policy for the current process (so you can execute the shellcode from the stack) (Vista SP1, XP SP3, Server 2008, and only when DEP Policy is set to OptIn or OptOut)
4) NtSetInformationProcess(). This function will change the DEP policy for the current process so you can execute your shellcode from the stack.
5) Virtual Protect(PAGE_READ_WRITE_EXECUTE). This function will change the access protection level of a given memory page, allowing you to mark the location where your shellcode resides as executable.
6) Write Process Memory(). This will allow you to copy your shellcode to another (executable) location, so you can jump to it and execute the shellcode. The target location must be writable and executable.[6]

4. Conclusion

This research paper deals with in-depth analysis about how vulnerability can be exploited in the hacker's point of view. It may even lead anyone possibly to end up with zero day vulnerability. Further research will focus upon how to bypass Microsoft's Enhanced Mitigation Exploit Toolkit (EMET), which corporate security widely uses it to avoid bypassing the above mentioned mitigations.

References

[1] "Buffer Overflow Attacks" by Ashley Hall and Huiming Yu, A&T University
[2] "The Concept of Dynamic Analysis", Thomas Ball, Bell Laboratories.

[3] "Data Execution Prevention", System and Network Analysis Center, IAD.
[4] "DEP/ASLR Implementation Progress in Popular Third Party Windows Application" by Alin Rad Pop, Secunia Research.
[5] "Proposal for Improvement of Implementation of SEHOP" by EMET
[6] "Corelan Exploit Developing", Corelan Team, USA.