Michal Bucko

sapheal.hack.pl
HACKPL Security Labs

Exploitation for phun and profit

The aim of the paper is to introduce the Readers with the currently used exploitation techniques. The paper does not cover the very basics of exploitation described in many papers before, however, it is not (yet?) a research paper into one of the fields of exploitation – the author wanted to write an overall review (not so basic) about the exploitation techniques. The aim of the research conducted was not to prepare a guideline throughout the exploitation techniques. The document is intented mainly for bug hunters and security specialists. The author is in no way responsible for any illegal actions caused by the Readers.

These days, bug hunters deal with various antidebugging, antidumping and antiexploitation mechanisms implemented in different systems. When building a working and reliable exploit, one must bypass different security mechanisms. DEP, Stackguard, ASLR are some of those. In this paper, I am going to show the ways of bypassing those security measures.

Must-know

STACK	Temporary local variables automatically alocated when the function is called
HEAP	Longer (and happier?) life Dynamic memory allocation and freeing
SANDBOXING	Stack buffer security-related checks heap-cookies (we have to eat something, don't we?)
EAT-MY-COOKIE	Memory allocation and freeing marking facility, missing or less-than-delicious cookies lead to exception-raise conditions ;-)
UEF	Unhandled Exception Filter
VEH	Vector Exception Handling, information about

	VEH is stored on the heap ;) _VECTORED_EXCEPTION_NODE structure
RtlEnterCriticalSection	Read about Process Environment Block and Thread Information Block (TIB/TEB) Overwriting the pointer to RtlEnterCriticalSection in PEB with our address

According to Microsoft ([1]), DEP (Data Execution Prevention) can be described the following way:

„*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. In Microsoft Windows XP Service Pack 2 (SP2) and Microsoft Windows XP Tablet PC Edition 2005, DEP is enforced by hardware and by software.*

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. „

DEP can be divided into hardware-enforced and software-enforced ones. When talking about hardware-enforced DEP ([1]):

„Hardware-enforced DEP marks all memory locations in a process as non-executable unless the location explicitly contains executable code. A class of attacks exists that tries to insert and run code from non-executable memory locations. DEP helps prevent these attacks by intercepting them and raising an exception. „

NX and XD features won't be described further in this paper. When talking about software-enforced one ([1]):

„An additional set of Data Execution Prevention security checks have been added to Windows XP SP2. These checks, known as software-enforced DEP, are designed to block malicious code that takes advantage of exception-handling mechanisms in Windows. 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. "

Maxpatrol once presented nice way of DEP and heap protection security measures bypass

([3])-

```
„If, during the overflow the concidental memory block is free and is residing in
the lookaside list, then it becomes possible to replace the Flink pointer with
an arbitrary value. Then, if the memory allocation of this block happens, the
replaced Flink pointer will be copied into the header of the lookaside list and
during the next allocation HeapAlloc() will return this fake pointer. „
```

The exemplary code snippet shown by Alexander Anisimov (Positive Technologies), which shows how to bypass heap protection measures, looks the following way:

```
mem1 = HeapAlloc(h, 0, 64-8);
printf("Heap block 1: %08X\n", mem1);

=
memset(mem1, 0x31, 64); BUFFER OVERLOW

memcpy((char *)mem1+64, "\x84\xFF\x12\x00", 4); FAKE ALLOCATION ADDRESS


mem2 = HeapAlloc(h, 0, 128-8); LOOKASIDE LIST ([3])
printf("Heap block 2: %08X\n", mem2);

mem3 = HeapAlloc(h, 0, 128-8);
printf("Heap block 3: %08X\n", mem3);
memset(shellcode, 0, sizeof(shellcode)-1);

memcpy(shellcode, "\x8B\xFF\x12\x00", 4); FAKE RET ADDRESS

memcpy(shellcode+4, "\x90\x90\x90\x90", 4); SHELLCODE ;-)
memcpy(shellcode+4+4, calc_code, sizeof(calc_code)-1);

memcpy(mem3, shellcode, sizeof(calc_code)-1+8); STACK FRAME OVERWRITE
```

[3] The exemplary heap protection bypass (lookaside-list-based)

DEP-bypass technique demonstrated in the very paper takes advantage of return-into-lib technique. Below, you can see how it works:

```
FAKE RETURN ADDRESS - system()

getaddr(&shellcode[0]);
memcpy(shellcode+4, "\x32\x32\x32\x32", 4);
```

Windows' security level has raised very much recently as Vista came into play. The use of ALSR in Vista (previously used in e.g. OpenBSD) has raised the bar higher (not to mention UAC). According to Michael Howard, an attacker has 1/256 chance of obtaining the right address (DLL or EXE file might be loaded into any of 256 locations). In one of my favourite security magazines (next to CBJ), *Informative Information for the Uninformed (uninformed.org, [2]),* bypassing of hardware-enforced DEP was quite thoroughly described. It could be bypassed in various ways, however, the addition of ASLR would make the exploitation much more difficult. ALSR's 8 bits of entropy (isn't entropy beautiful, automagically everything works if you apply the rules) definetely helps to mitigate the attacks with the hardcoded addresses. In order to bypass ASLR, we usually use heap spraying technique. ASLR can also be bypassed using LSB-overwrites.

heap-spray-phun

The technique is commonly used in various exploits, it was first shown by SkyLined (MS04-040, MS05-020). It is mainly used when (typically, a browser ;-)) an application call/jmp into invalid memory (within the possible heap range address and lower than 0x7fffffff). The technique takes advantage of heap-inject of NOP-shellcode pairs.

Too typical combination

Typical combination: ASLR and DEP can be bypassed (nicely shown by Alex Sotirov, ANI exploit) by bruteforce attack targeted at location of DEP disable in NTDLL. The exploit itself takes a jump to DEP-disabling code. Finally, the payload gets executed. There is no need to find shellcode's location as heap spraying is used.

return-to-libc

Bypassing non-executable stack can also be conducted using return-to-libc method. The aforementioned method takes advantage of library functions. The ret-address gets overwritten, however, with one of the functions from libc. As the very functions don't reside on the stack, we can bypass the stack protection. The detailed description of this method can be found in various papers – I recommend c0ntex's papers. There are various ways of return-to-libc prevention, new ways of exploitation are used. For instance, sometimes it is possible to brute force the address of libc function (this is not advised due to log entries).

return-to-got

As you probably know, Global Offset Table stores absolute locations of function calls. Overwriting the entry in GOT allows us to redirect the application flow – we don't overwrite the next instruction with shellcode's address and simply patch the GOT reference with a function that we use to run system commands. I assume that neither return-to-libc, nor return-to-GOT techniques must be described thouroughly as such description might be found in various different papers on non-executable stack bypass techniques

SEH-overwrite

In the paper, I assume that you fully understand the technique itself. We already understand that we can possibly try to overwrite the pointer to the exception handler with an address outside the address range of a loaded module. Elaboration on SEH-overwrite-based exploitation's prevention touches the issue of supplying the value for the handler of ERR (exception registration record). /SAFESEH-compilation is secure only if all of the images loaded into the address space have been compiled with /SAFESEH. The design of SEH-exploitation prevention would involve exception interception (before it is passed to Registered Exception Handler). According to skape, the interception would take place here: ntdll!RtlDispatchException. For further information, please, refer to [14].

(quite) automated ActiveX controls exploitation

There are many ActiveX controls available, there is real and working certification of such controls. The problem appears to be very serious as combination of attacks might lead unaware users to dowload and use various malicious ActiveX controls. Moreover, there are also various ActiveX controls with kill-bit set, which could be used by malware (already installed) to make the victim's machine even more vulnerable. For instance, the simplest SEH-overwrite-based exploit of the ActiveX control is shown below:

```
<html>

<body>
<OBJECT id="target" WIDTH=445 HEIGHT=40 classid="clsid:..." > </OBJECT>

<script language="vbscript">

shellcode=unescape(...) SHELLCODE
nop=unescape("%90..”)    NOP-SLED
pointer_to_seh=unescape("..") POINTER TO SEH
seh_handler=unescape("..") SEH HANDLER

arg=String(3256,"A")    BUFFER OF 41's ;-)

arg1=arg+pointer_to_seh+seh_handler+nop+shellcode+nop OUR ARGUMENT

target.VulnerableFunction arg1 FUNCTION EXECUTION

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

It is also simple to write an EIP-overwrite exploit template for ActiveX control-based vulnerabilities, which is going to result in appearing of tons of such exploits. The problem is quite obvious – such add-on facilities should be certified. Users should be advised not to install unknown ActiveX controls (without a certificate).

Flaw development

Current vulnerability development process touches various aspects of security analysis: static and dynamic code analysis, file format fuzzing, protocol fuzzing, etc. One of the most interesting aspects of flaw development is connected with so-called code coverage. Code coverage is a term describing the measure of exercised code within the tested application. Various techniques are used to increase the probability of detecting the flaw. CFG (control flow graphs), IDC scripts, IDA Python, PaiMei, various fuzzing tools, such as Peach Framework might be used.

Research proposals

There are many various ways of tackling exploitation. One of those is to analyze the memory access behaviour – we could do this using DBI (dynamic binary instrumentation), PAI (page access interception) or NULL segment interception – all those are described by skape in his paper [9]. The following techniques could be used (mainly for) memory access isolation and data propagation. The other way of tackling advanced exploitation techniques is to involve external devices that would help to raise the bar in the field of data execution prevention (especially, randomization!). Such devices could be developed using FPGA. The other technique would involve a combination of layering system and ACL (virtualization) – the weakest link of such system would be placed between the operating system and a virtual machine. The anomaly detection based on neural networks might also become very important in the future as evasion techniques evolve.

Finally, please, feel free to correct me if I am wrong in any part of this document. I would be grateful if you could also could provide interesting information, which could be helpful while providing another pentest/vulnerability assessment research. I am currently running various groups conducting various research works, please, feel free and contact me if you want to join us – we do different works, from microcontrollers , through telecommunications till security. My address:

Michal Bucko

sapheal@hack.pl
http://sapheal.hack.pl
HACKPL Security Laboratories

- - - - - - - - - - - - - - - - - - this does not belong to the article - - - - - - - - - - -

According to „Ksiega krolewska", in order to understand one should be able to lose everything and leave, one should not concentrate on the simple recognition and science, one should lie himself between those who are defeated. But, those are only the words, misinterpretation will touch them inevitably..

Literature:

[1] A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003 (http://support.microsoft.com/kb/875352)
[2] Bypassing Windows hardware-enforced DEP (http://www.uninformed.org/?v=2&a=4&t=pdf)
[3] Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass (http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf)
[4] Windows Heap Overflows , David Litchfield, NGSSoftware (BH, http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt)
[5] Return-to-libc (http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf), c0ntex
[6] How to hijack the Global Offset Table with pointers for root shells (http://www.infosecwriters.com/text_resources/pdf/GOT_Hijack.pdf), c0ntex
[7] Automated analysis research document, Eleytt Interim Research, HACKPL [no reference]
[8] Automated malware analysis, Eleytt Interim Research, HACKPL [no reference]
[9] Memalyze: Dynamic Analysis of Memory Access Behavior in Software (http://www.uninformed.org/?v=7&a=1)
[10] (http://pedram.redhive.com/PaiMei/docs/), PaiMei, Pedram Amini
[11] Raising the bar for rootkit detection; Sparks, Bulter, Shadow Walker (https://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf)
[12] Memory Access Detection, Horovitz, Oded (http://cansecwest.com/core03/mad.zip)
[13] Reducing the entropy for GS Cookies (http://www.uninformed.org/?v=7&a=2)
[14] Preventing the exploitation of SEH overwrites, skape (http://www.uninformed.org/?v=5&a=2)
[15] Practical SEH exploitation, Johnny Cyberpunk, The Hacker's Choice (http://thc.org/papers/Practical-SEH-exploitation.pdf)
[16] Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server, David Litchfield , NGSSoftware (BH, http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf)
[17] Effective Bug Discovery, vf (http://uninformed.org/?v=5&a=5)