

SRS DAY

Introduction to stack protections in Windows Vista

Julien Bachmann

Pierre-Olivier Martel

Introduction

Microsoft's marketing strategy for Vista was his security. This operating system is a target loved by hackers as it represents nearly 93% of the market. For those three reasons, we thought it is interesting to take a closer look at Windows Vista's stack protections.

This paper will only deal with stack protections. They do not all come with Vista, but it is in this version of the operating system that they are all used. It is also since Windows Vista that the protections became mandatory for programs compiled with Visual Studio 2005.

Overview of buffer overflow

In order to exploit a program, his execution flow will be changed. This is done by injecting a shellcode that will be executed instead of the original instructions.

A shellcode is an executable code which is position-independent who should be executed in another program's context. It is represented by a string filled with op-codes.

This shellcode will be injected into the target mostly by a buffer overflow. It means that a buffer located in the stack is used in order to change the return address which is also stored in the stack. This kind of vulnerability is often found when *strcpy()* is used without checking the size of the data copied on the stack.

Compilation-time protection

One of the first things a programmer will see is the new functions of the CRT (C RunTime library). Since recently, some of the functions became forbidden and are replaced by most secure ones. Those functions belong to StrSafe and safeCRT libraries.

Functions from those two libraries add tests before operations on strings in order to avoid buffer overflow. For example:

```
void Function(char *s1, char *s2) {
    char temp[32];
    strcpy(temp, s1);
    strcat(temp, s2);
}
```

Will act like:

```
HRESULT Function(char *s1, char *s2) {
    char temp[32];
    HRESULT hr = StringCchCopy(temp, sizeof(temp), s1);
    if (FAILED(hr)) return hr;
    return StringCchCat(temp, sizeof(temp), s2);
}
```

The call to *strcpy()* was replaced by a call to *StrCchCopy()* which takes an additional parameter: the length of the destination buffer.

Canaries (/GS)

Canaries or Canary Words are known values that are placed between a buffer and control data on the stack to monitor buffer overflows. When the buffer overflows, it clobbers the canary, making the overflow evident. The creation of a Canary Word can be reproduced with the following function:

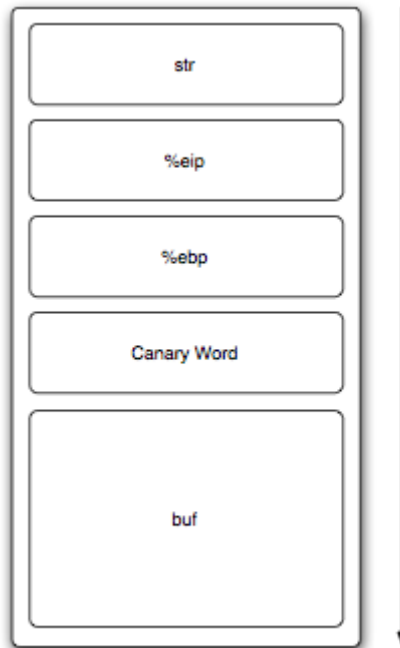
```
int main()
{
    FILETIME ft;
    unsigned int    cookie = 0;
    unsigned int    tmp = 0;
    unsigned int*   ptr = NULL;
    LARGE_INTEGER   perfcnt;

    GetSystemTimeAsFileTime(&ft);
    cookie = ft.dwHighDateTime ^ ft.dwLowTimeDateTime;
    cookie = cookie ^ GetCurrentProcessId();
    cookie = cookie ^ GetCurrentThreadId();
    cookie = cookie ^ GetTickCount();

    QueryPerformanceCounter(&perfcnt);
    ptr = (unsigned int)&perfcnt;
    tmp = *(ptr + 1) ^ *ptr;
    cookie = cookie ^ tmp;
}
```

The following prelude is appended to each function containing a local static buffer:

```
00401006 MOV EAX,DWORD PTR DS:[__security_cookie]
0040100B XOR EAX,ESP
0040100D MOV DWORD PTR SS:[ESP+400],EAX
```

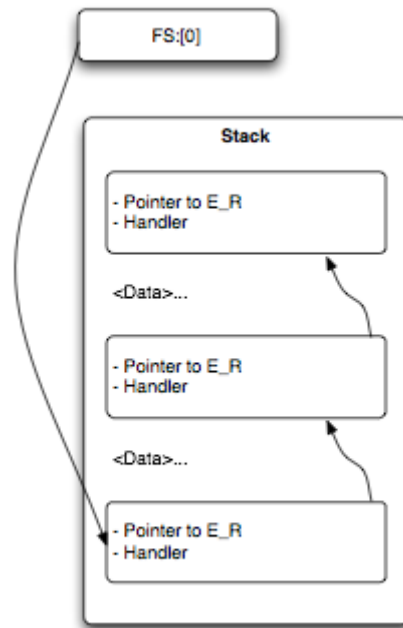


Exceptions handling (SEH)

Overview

Since Windows 2003, exceptions are handled by a method named SEH (structured Exception Handling). An exception handler will be called in situations like a wrong memory access or a division by zero.

Each function will push an EXCEPTION_REGISTRATION structure on the stack. This structure contains a pointer to the preceding element of the linked list and a pointer to the error handler. All the elements are stored in the stack and the head could be reached through FS: [0]. When an exception will occur, the operating system will go through the list and call the first valid handler.



A safer solution: safeSEH

All the elements are stored in the stack and will be called if an exception occurs; it is possible to change the program execution flow. If the entire stack is rewritten higher than its start address, an exception will be generated and the canary verification will be bypassed.

After Microsoft's security engineers saw this problem, they rewrote parts of the exception handling mechanism. The new mechanism called safeSEH is mandatory since Windows XP SP2 for all the programs compiled with Visual Studio.

When safeSEH is used, the operating system will run some tests before calling the handler. A handler will only be called if:

- He is not in the stack ;
- He belongs to another process or library and is considered safe by this library ;
- He is in the heap.

Despite those tests, a workaround still exists in the implementation of safeSEH. If the address of the handler belongs to a memory mapped file, the call will be made without any other tests than the one from the stack.

The case with the buffer in the heap is very rare, that is why it is not described in this paper. Indeed, it is code specific.

Using safeSEH to bypass canaries

As said before, the program execution flow could be modified by rewriting the nearest EXCEPTION_REGISTRATION and writing beyond the stack limit. In order to bypass the canary test, those steps should be followed:

- Write the shellcode in the stack ;

- Fill the stack until nearest EXCEPTION_REGISTRATION structure is reached ;
- Modify the structure in order to call the shellcode ;
- Rewrite the stack beyond his limit in order to generate an exception.

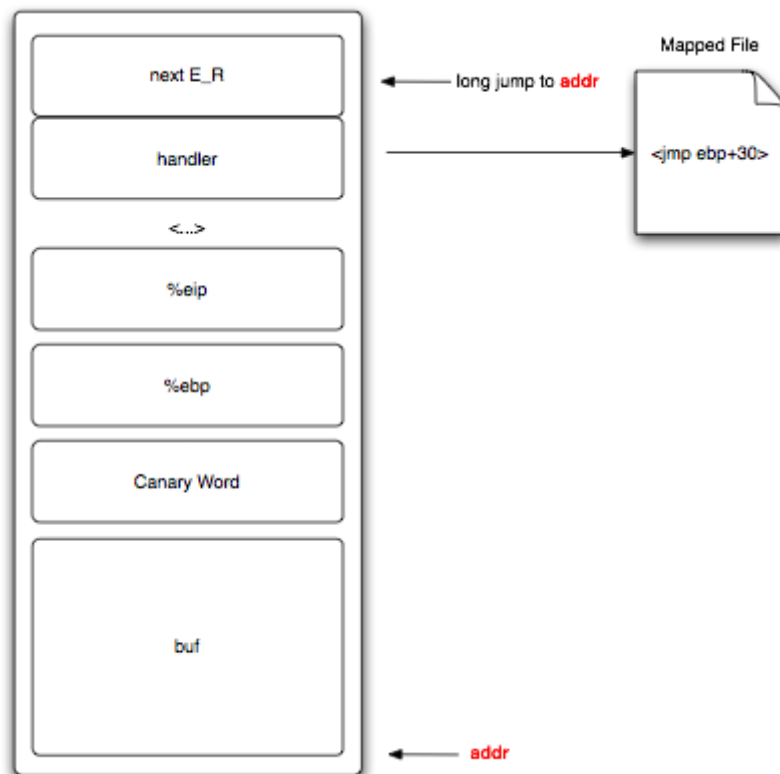
David Lichtfield studied the behavior of the registers while an exception was handled. He demonstrated that a lot of pointers are going to the EXCEPTION_REGISTRATION that was rewritten during the attack. His result could be found in the following table:

ESP	+8	+14	+1C	+2C	+44	+50
EBP	+0C	+24	+30	-04	-0C	-18

If the attacker finds an instruction that is safe for the operating system and that looks like one of the following instructions, he will be able to jump on the second part of the EXCEPTION_REGISTRATION record.

- Call dword ptr[esp + NN]
- Call dword ptr[ebp + NN]
- Call dword ptr[ebp - NN]
- Jmp dword ptr[esp + NN]
- Jmp dword ptr[ebp + NN]
- Jmp dword ptr[ebp - NN]

As this record is also in the stack, the attacker can write in it an instruction that will jump to his shellcode. The distance between the EXCEPTION_REGISTRATION record and the beginning of the shellcode buffer should be easy to find with a debugger.



Address Space Layout Randomization (/dynamicbase)

Address space layout randomization aims to protect against some types of security attacks by preventing an attacker to predict target addresses. The goal of this technology is to introduce randomness into addresses used by a given task. The randomization is more effective when you have good entropy in the random offsets.

Thus, an attacker needs to guess efficiently the position of each memory area he wants to attack. It is even often mandatory for him to guess the right address at the first attempt because failed attempts will most likely crash the attacked task.

For the time being, ASLR is implemented in OpenBSD, Pax and ExecShield for GNU/Linux, Gentoo Hardened and several releases of RedHat Enterprise. Microsoft engineering made the integration in Windows Vista Beta 2 for the binaries which are linked to be ASLR-compliant (/dynamicbase). To see exactly what ASLR is, we will take the example of a GNU/Linux system patched with PaX extensions. We first copy /bin/cat into /tmp/cat and we disable the PaX ASLR protection. Then, we run the following command:

```
/tmp/cat /proc/self/maps
```

We obtain the following result:

```
[1] 08048000-0804a000 R+Xp 00000000 00:0b 812 /tmp/cat
[2] 40000000-40015000 R+Xp 00000000 03:07 110818 /lib/ld-2.2.5.so
[3] 4001e000-40143000 R+Xp 00000000 03:07 106687 /lib/libc-2.2.5.so
[4] bffffe000-c0000000 RWXp fffff000 00:00 0
```

[1] Corresponds to the code and data ELF segments of /tmp/cat.

[2] Corresponds to the dynamic linker.

[3] Corresponds to the libc library.

[4] Corresponds to the stack growing downwards.

There are several other mappings not shown in this example: The heap managed by brk(), following [2] and the other mappings that the task can create with mmap(). These mappings will be located between [3] and [4] unless the MAP_FIXED flag is passed to mmap(). For our purposes, the mappings will be grouped in three main groups:

- [1], and the heap managed by brk()
- [2], [3] and the mappings created by mmap()
- [4], the stack

The areas of the first and the last groups are created during the address space creation and are fixed (only the size can change), whereas the mappings in the second group may come and go during the lifetime of the task. Since the base addresses of each group are not linked, it is possible to randomize them independently. Then, if an attacker successfully predicts an address, the other groups stay safe. Moreover, if a technique requires advance knowledge of base addresses from several groups, the

2	0.25	0.68	0.99	≈1	≈1	≈1	≈1	≈1	≈1	≈1	≈1	≈1	≈1	≈1
4	0.06	0.23	0.64	0.98	≈1	≈1	≈1	≈1	≈1	≈1	≈1	≈1	≈1	≈1
8	≈0	0.02	0.06	0.22	0.63	0.98	≈1	≈1	≈1	≈1	≈1	≈1	≈1	≈1
16	≈0	≈0	≈0	≈0	≈0	0.02	0.22	0.98	≈1	≈1	≈1	≈1	≈1	≈1
24	≈0	≈0	≈0	≈0	≈0	≈0	≈0	0.02	0.06	0.63	≈1	≈1	≈1	≈1
32	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	0.63	≈1	≈1	≈1
40	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	0.63	≈1	≈1
56	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	0.63	≈1

Pb(x)	x													
N	1	4	16	64	256	2^10	2^14	2^18	2^20	2^24	2^32	2^40	2^56	2^64
1	0.50													
2	0.25	1												
4	0.06	0.25	1											
8	≈0	0.02	0.06	0.25	1									
16	≈0	≈0	≈0	≈0	≈0	0.02	0.25							
24	≈0	≈0	≈0	≈0	≈0	≈0	≈0	0.02	0.06	1				
32	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	1			
40	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	1		
56	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	1	

It is obvious that from the defense point of view, the goal is to make N as high as possible, while keeping x as low as possible. We do not have any control on N (it is not feasible to re-randomize the address space layout during the runtime). On the other hand, we control x because when the attacker makes a wrong guess, the application will go into a state that will likely result in a crash. Then, it would be a good start to setup a crash detection system coupled with ASLR.

Attacking ASLR

Several types of attacks can be used to decrease the entropy available for the address space layout. A string format bug type can be used to extract the return address or the frame pointer of a function, revealing the address of a vulnerable library. Some systems also randomize the dynamic library loading order to protect against these types of attacks.

It is also possible to specifically decrease the heap or stack randomness. The stack is typically aligned to 16 bytes, which minimizes the randomness range. In the same way, the heap is aligned to a memory page (typically 4096 bytes).

Currently, few researches have been achieved on the different ASLR implementations. For example, it is impossible to predict that you cannot use a stack-overflow or heap-overflow type attack to obtain some information about the randomization. Moreover, it is unknown how many bugs that have been previously neglected can be used against ASLR (that is, bugs giving a read-only – or write – access to the attacker).

ASLR may imply two other negative side effects. The randomization process moves large memory areas, independently. Consequently, it also modifies the gaps between these areas. These changes imply a random modification in the mappings maximum size (heap, stack, ...). Applications expecting to use these memory spaces may not be able to be run. In a second time, since each address space

creation needs several random bits, ASLR increases the use of the system entropy generator. On the other hand, a cryptographic-qualified entropy generator is not mandatory.

Conclusion

In this article, we showed that it is still possible to exploit a buffer overflow. Anyway, Microsoft's researchers have done a good work that will slow down most the attacks.

With those technologies, they leveled up the operating system's security, but as every system it is not invulnerable.

References

The Shellcoder's Handbook (Wiley), Chapter 8: « Windows Overflows »

David Lichtfield, "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server"

PaX : <http://pax.grsecurity.net/docs/aslr.txt>

Phrack, "Bypassing PaX ASLR protection",
<http://www.phrack.org/archives/59/p59-0x09.txt>

Wikipedia, Address Space Layout Randomization,
http://en.wikipedia.org/wiki/Address_space_layout_randomization

Michael Howard, "Address Space Layout Randomization in Windows Vista",
http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx

Michael Howard, "Alleged Bugs in Windows Vista's ASLR Implementation",
http://blogs.msdn.com/michael_howard/archive/2006/10/04/Alleged-Bugs-in-Windows-Vista-1920-s-ASLR-Implementation.aspx