SYSDREAM

IT Security Services

Bypassing SEHOP

Stéfan Le Berre
s.leberre@sysdream.com

Damien Cauquil
d.cauquil@sysdream.com

Table of contents

0. Introduction

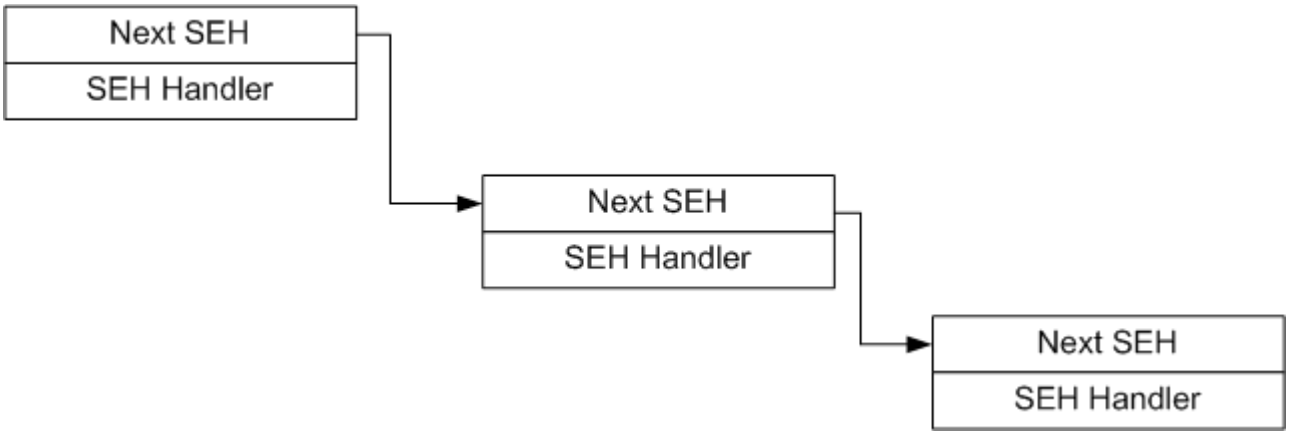
Microsoft has recently implemented in many Windows versions a new security feature named « Structured Exception Handling Overwrite Protection » **[1 & 2]**. Those systems are:

- Microsoft Windows 2008 SP0
- Microsoft Windows Vista SP1
- Microsoft Windows 7

We did not found any known attack aiming at defeating this new feature but only many papers describing the feature itself and its robustness. Indeed, SEHOP seems to be so reliable that Microsoft released a patch in order to activate this security feature by default on all programs. Is it already the end of stack overflows under Microsoft Windows? Not yet, but under some circumstances, as we will explain below.

1. SEHOP specifications (short version)

SEHOP is an extension of Structured Exception Handling and implements more security checks on SEH structures used by programs. The core feature of SEHOP checks the chaining of all SEH structures present on the process stack and especially the last one, which should have a special handler value pointing right onto a function located in ntdll. Here is a classical SEH chain:



Each SEH structure points to the next structure and the last one contains a specific handler pointing to ntdll!_except_handler4. When exploiting by overwriting a given SEH structure onto the stack, the next SEH pointer is overwritten in order to contain some bytecode and the SEH handler is overwritten to point to a sequence of « POP POP RET » instructions located in a non-SafeSEH module.

The validation algorithm used in SEHOP has been exposed by *A. Sotirov* during Black Hat in year 2008 **[3]**. Let's have a look at it:

```
BOOL RtlIsValidHandler(handler)
{
  if (handler is in an image) {
     if (image has the IMAGE_DLLCHARACTERISTICS_NO_SEH flag set)
        return FALSE;
     if (image has a SafeSEH table)
        if (handler found in the table)
           return TRUE;
        else
           return FALSE;
     if (image is a .NET assembly with the ILonly flag set)
        return FALSE;
     // fall through
  }
  if (handler is on a non-executable page) {
     if (ExecuteDispatchEnable bit set in the process flags)
        return TRUE;
     else
        // enforce DEP even if we have no hardware NX
        raise ACCESS_VIOLATION;
  }
  if (handler is not in an image) {
     if (ImageDispatchEnable bit set in the process flags)
        return TRUE;
     else
        return FALSE; // don't allow handlers outside of images
  }
  // everything else is allowed
  return TRUE;
}

[...]

// Skip the chain validation if the
DisableExceptionChainValidation bit is set
if (process_flags & 0x40 == 0) {
  // Skip the validation if there are no SEH records on the
  // linked list
  if (record != 0xFFFFFFFF) {
     // Walk the SEH linked list
     do {
        // The record must be on the stack
        if (record < stack_bottom || record > stack_top)
           goto corruption;
        // The end of the record must be on the stack
        if ((char*)record + sizeof(EXCEPTION_REGISTRATION) >
stack_top)
           goto corruption;
        // The record must be 4 byte aligned
        if ((record & 3) != 0)
```

```
         goto corruption;
      handler = record->handler;
      // The handler must not be on the stack
      if (handler >= stack_bottom && handler < stack_top)
         goto corruption;
      record = record->next;
   } while (record != 0xFFFFFFFF);

   // End of chain reached
   // Is bit 9 set in the TEB->SameTebFlags field?
   // This bit is set in ntdll!RtlInitializeExceptionChain,
   // which registers FinalExceptionHandler as an SEH handler
   // when a new thread starts.
   if ((TEB->word_at_offset_0xFCA & 0x200) != 0) {

      // The final handler must be ntdll!FinalExceptionHandler
      if (handler != &FinalExceptionHandler)
         goto corruption;
   }
  }
}
```

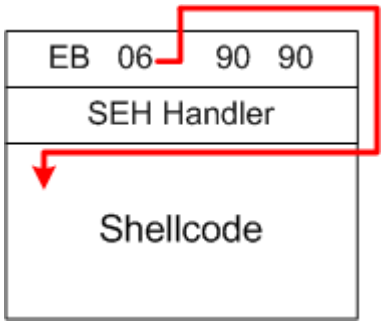
We have to take in consideration some constraints:

- SEH handler should point onto a non-SafeSEH module
- The page should be executable
- The SEH chain should not be altered and must end with a SEH structure containing a special value (0xFFFFFFFF as next SEH structure pointer and a specific value as SEH handler)
- All SEH structures should be 4-byte aligned
- Last SEH structure's handler should point right into ntdll to ntdll!FinalExceptionHandler routine
- All SEH pointers should point to stack locations

2. Dealing with SEHOP when exploiting a stack overflow

2.1. Breaking out the classical exploitation scheme

Here is the classical exploitation scheme used:



The SEH handler points to a « POP POP RET » instruction sequence, and the first member of the structure contains code instead of a valid stack address. While handling an exception, Windows transfers control to exception handlers, following the SEH chain. Our first exception handler has been overwritten and the execution flow is redirected onto the « POP POP RET » sequence (instead of really handling the exception). This sequence transfers the execution to the first two bytes of the first member of the current SEH structure, which is a jump instruction (0xEB06) to our shellcode.
By doing this, the SEH chain has broke up:

The first two bytes of the first DWORD of the SEH structure are never executed and could be altered in order to make the DWORD a valid stack address. It is also possible to alter the other two bytes to make the DWORD point to a valid stack address we control and to define those two bytes as a jump instruction. The DWORD can be evaluated both as a valid stack address and a valid instruction doing a jump when called.
If we put some fake SEH structure at stack address pointed by this first DWORD, we can fake an entire SEH chain and control the second SEH structure referenced by the first one in order to make it valid.

2.2. The tricky part

An issue remains: which jump instruction could be coded with a bytecode representing a 4-byte aligned address? Only one instruction fits here: JE (coded as 0x74). This instruction is a conditional jump: the jump is done only if Z flag is set. Z flag is not set by default when Windows handles an exception, we have to set it by executing an instruction computing a zero value for instance. The « Xor » operator seems to be a good choice, and our solution is then clear: we have to jump onto a « XOR, POP, POP, RET » sequence in order to interpret the JE instruction as a JMP. This was the tricky part of this SEHOP bypass technique.

« XOR, POP, POP, RET » instructions are not so difficult to find, we can find some sequences by looking at end of functions returning a null result:

```
XOR EAX, EAX
POP ESI
POP EBP
RET
```

In order to ensure the exploitation, we put in our Proof of Concept a « XOR, POP, POP, RET » sequence.

Say we have a first SEH structure at 0x0022FD48, we could create a second one at 0x0022FD74 containing a next SEH pointer with a value of 0xFFFFFFFF and handler pointing to ntdll! FinalExceptHandler. If we set the first SEH structure's handler to point onto a « XOR, POP, POP, RET » sequence, we can redirect the execution flow to whatever we want without really breaking the chain. Concretely, we recreate a valid fake SEH chain that redirects the execution onto a specific shellcode.

A major limitation resides in the ASLR feature present in Microsoft Windows 7 and Vista. The exploitation relies on the fact we know the address of ntdll!FinalExceptHandler but in reality this address changes at each reboot of the target machine. Ntdll's ImageBase is randomized at boot time and makes exploitation harder. We made some tests onto the ASLR (without reversing it) and it seems that only 9 bits on the 16 representing the ImageBase are randomized, which represents 1 chance over 512 to successfully exploit a stack overflow with SEHOP bypass.

3. Proof Of Concept

3.1. Target program & constraints

We created a little program to demonstrate our technique on Windows 7. This program just copies the content of a file (« OwnMe.txt ») into memory and smashes the stack during this operation, thus raising an exception caught by the exception handler.

All we have to do is:

- create a valid fake SEH chain
- set the last SEH structure's handler to point to ntdll!FinalExceptionHandler
- put a shellcode onto the stack (Windows 7 compliant)
- locate a « XOR, POP, POP, RET » sequence into memory

First of all, the « XOR, POP, POP, RET » sequence is known because we put it in the program's code. We can find this sequence at memory address 0x004018E1.

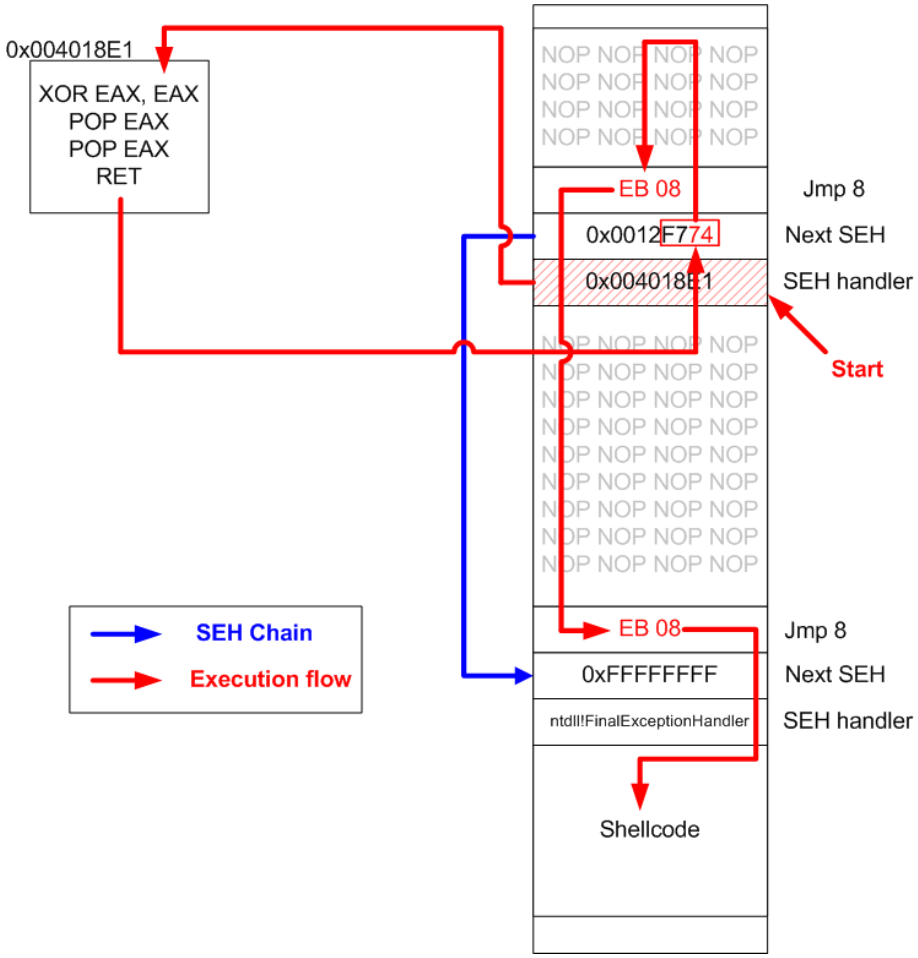
3.2. crash and exploitation

When program crashes, we have:

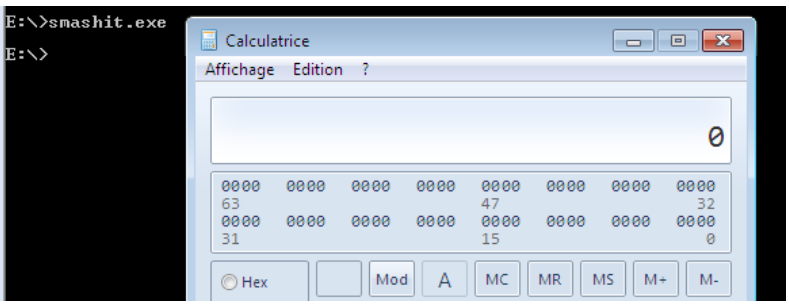
```
0012F700   41414141  Pointer to next SEH record
0012F704   41414141  SE handler
```

Then we need to create a valid fake SEH chain with a second SEH structure stored at 0x0012F774 containing 0xFFFFFFFF as first member of the structure (next SEH pointer) and ntdll! FinalExceptionHandler as SEH handler. SEH structure located at 0x0012F700 is then modified in order to point to the fake SEH structure we have just created, and the handler is set to 0x004018E1. A jump instruction (JMP +8) is placed before each SEH structure in order to avoid data execution.

When exploiting, execution should work as following:



This shellcode will only launch calc.exe:



NB: The target program and a working exploit are available in the zip joined to this white paper. You can also download it from Sysdream's website **[4].**

4. Conclusion

SEHOP is not the ultimate protection against stack overflows if used alone. Since this SafeSEH extension was released, too many people consider it as unbreakable. We just demonstrated that it is possible to bypass the Structured Exception Handling Overwrite Protection under some circumstances.

But SEHOP is an excellent native security feature when used in conjunction with ASLR and DEP, we cannot deny it.

5. Credits

http://sysdream.com/
http://ghostsinthestack.org/
http://virtualabs.fr/

6. Bibliography

[1] Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP: http://blogs.technet.com/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx

[2] SEHOP per-process opt-in support in Windows 7: http://blogs.technet.com/srd/archive/2009/11/20/sehop-per-process-opt-in-support-in-windows-7.aspx

[3] Bypassing Browser Memory Protections: http://taossa.com/archive/bh08sotirovdowd.pdf

[4] ZIP containing our target program & exploit http://www.sysdream.com/SEHOP.zip