A Brief History of Exploitation Techniques & Mitigations on Windows

By Matt Miller
Agenda

• Introduction
  – What are exploit mitigations?

• Evolution of exploit mitigations on Windows
  – /GS, SafeSEH, DEP, ASLR

• A look toward the future
Overview

• Software vulnerabilities are common

• Reliable exploitation techniques exist
  – Stack-based buffer overflows
  – Heap overflows (not covered due to time)

• Exploit mitigations act as countermeasures to these techniques
What are exploit mitigations?

• Prevent or impede exploitation

• Patching the vulnerability
  – The only guaranteed mitigation (if done right)

• Workarounds
  – Disabling the vulnerable service

• Generic mitigations
  – Buffer overflow prevention
Exploit techniques & mitigations

THE LOGICAL EVOLUTION
Starting from the beginning

Common structure of an x86 stack frame

Local Variables | Saved EBP | Saved EIP | Arguments

Stack grows toward lower addresses
Exploit: Overwrite saved EIP

- Common stack-based buffer overflow\(^7\)
- Return address is overwritten with address of shellcode
Mitigation: Stack canaries (/GS)

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>GS Cookie</th>
<th>Saved EBP</th>
<th>Saved EIP</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xa47c10390x0012ef040x7601148c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x414141410x414141410x7843110b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Compiler change introduced in VS2002[7]
- Canary is validated before a function returns
- Mismatching canary leads to process termination
Exploit: Overwrite variables

```c
void vulnerable(char *in, char *out) {
    char buf[256];
    strcpy(buf, in);    // overflow!
    strcpy(out, buf);   // out is corrupt
    return;            // canary checked
}
```

- Canary is only checked at function return
- Corrupt arguments or locals may be used before return
- Attacker could overwrite canary or other memory \[2, 8\]
Mitigation: /GS improvements

• “Safe” copies of arguments made as locals

• Arrays positioned directly adjacent to GS cookie

• Corruption of dangerous locals and arguments is less likely
Exploit: SEH Overwrite

- **Structured Exception Handler (SEH) overwrite** [1]
  - **Handler** overwritten during overflow
  - Called when an exception is generated

- Exception can be generated before the canary is checked

```c
void vulnerable(char *ptr){
    char buf[128];
    try {
        strcpy(buf, ptr);
        ... exception ...
    } except(...) { }
}
```
Exploit: SEH Overwrite (cont’d)

Normal SEH Chain

N H → app!_except_handler4
N H → k32!_except_handler4
N H → ntdll!_except_handler4
0xffffffff

Corrupt SEH Chain

N H → 0x7c1408ac
0x414106eb

An exception will cause 0x7c1408ac to be called as an exception handler as:

```c
EXCEPTION_DISPOSITION Handler(PEXCEPTION_RECORD Exception,
    PVOID EstablisherFrame,
    PCONTEXT ContextRecord,
    PVOID DispatcherContext);
```
Mitigation: SafeSEH

- VS2003 compiler change (/SafeSEH) [9]
- Binaries are compiled with a table of safe exception handlers
- Exception dispatcher checks if handlers are safe before calling

Valid

Invalid SEH Handler

app!_main+0x1c

Safe SEH Handler

app!_except_handler4

app!eh1

app!eh2

app!_except_handler4

...
Exploit: SEH Overwrite Part II

- SafeSEH only works if all binaries in a process are compiled with it [4]

- Handler can be pointed into a binary that does not have a safe exception handler table
Mitigation: Dynamic SafeSEH

- Dynamic protection against SEH overwrites [4]
  - No compile time hints required

- Symbolic Validation frame inserted as final entry in chain

- Corrupt Next pointers prevent traversal to validation frame

Valid SEH Chain

Invalid SEH Chain

Can’t reach validation frame!
Recap: GS and SafeSEH

• GS and SafeSEH are solid mitigations for stack-based buffer overflows

• Applications must be recompiled
  – With the exception of dynamic SafeSEH

• Additional runtime mitigations are needed
  – Protection for legacy & 3rd party applications
Mitigation: Hardware DEP (NX)

- Exploits typically attempt to run shellcode stored in writable memory regions [10]
- Enforcing non-executable pages prevents execution of arbitrary shellcode
  - Binary must indicate support, VS2005 sets flag
Exploit: ret2libc

- NX stack and heap prevents arbitrary code execution
- Library code is executable and can be abused[11]
- Example: return into a library function with a fake call frame
Exploit: ret2libc (cont’d)

<table>
<thead>
<tr>
<th>Address of VirtualProtect</th>
<th>Address of jmp esp</th>
<th>Address of shellcode</th>
<th>Size of shellcode</th>
<th>RWX</th>
<th>Writable address</th>
<th>shellcode</th>
</tr>
</thead>
</table>

1. Return from vulnerable function
2. Entry to VirtualProtect
3. Return from VirtualProtect

- Windows makes extensive use of **stdcall**
  - Caller pushes arguments
  - Callee pops arguments with **retn**

- Allows multiple functions to be chained in ret2libc
Exploit: ret2libc (cont’d)

• Returning to `VirtualProtect` requires the ability to use NULL bytes
  – Often impossible (string-related overflows)

• Windows has an API to disable NX for an entire process
  – `NtSetInformationProcess[0x22]`

• `ntdll` calls this API & we can abuse it[3]
Exploit: NtSetInformationProcess

<table>
<thead>
<tr>
<th>Address of NtdllOkayToLockRoutine (0x7c952080)</th>
<th>$n$ byte pad</th>
<th>Address Of LdrpCheckNXCompatibility+0x13 (0x7c91d3f8)</th>
<th>4 byte pad</th>
<th>Address of jmp esp (0x1b4c7814)</th>
<th>$z+4$ byte pad</th>
<th>shl code</th>
</tr>
</thead>
</table>

ESP

app\!vulnerable+0x1c:
104713a4 c20400 retn 4 \(\leftarrow\) **Return to** NtdllOkayToLockRoutine and **add 4 to esp** \((n=4)\)
Exploit: NtSetInformationProcess

<table>
<thead>
<tr>
<th>Address Of</th>
<th>4 byte pad</th>
<th>Address Of</th>
<th>z+4 byte pad</th>
<th>shl code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LdrpCheckNXCompatibility+0x13 (0x7c91d3f8)</td>
<td></td>
<td>jmp esp (0x1b4c7814)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ESP

ntdll!NtdllOkayToLockRoutine:

7c952080 b001 mov al, 0x1  ← Set al to 1
7c952082 c20400 ret 0x4  ← Return to

LdrpCheckNxCompatibility+0x13
Exploit: NtSetInformationProcess

Address of jmp esp (0x1b4c7814)  
\[ z+4 byte pad \] \[ shl code \]

ntdll!LdrpCheckNXCompatibility+0x13:

7c91d3f8 3c01 cmp al, 0x1 \( \leftarrow \) al is equal to 1
7c91d3fa 6a02 push 0x2 \( \leftarrow \) Set esi to 2
7c91d3fc 5e pop esi
7c91d3fd 0f84b72a0200 je 7c93feba \( \leftarrow \) ZF=1, jump

...  

7c93feba 8975fc mov [ebp-0x4], esi \( \leftarrow \) Set [ebp-4] to 0x2
7c93feb0 e941d5fdff jmp 7c91d403

...  

7c91d403 837dfc00 cmp [ebp-0x4], 0x0 \( \leftarrow \) [ebp-4] is not 0
7c91d407 0f8560890100 jne 7c935d6d \( \leftarrow \) ZF=0, jump
Exploit: NtSetInformationProcess

ZwSetInformationProcess(
    NtCurrentProcess(),
    ProcessExecuteFlags,
    &ExecuteFlags,
    sizeof(ULONG));

Address of
jmp esp
(0x1b4c7814)

ESP

7c935d6d  6a04  push 0x4  \(\leftarrow\) Length := 4
7c935d6f  8d45fc  lea eax, [ebp-0x4]
7c935d72  50  push eax  \(\leftarrow\) &[ebp-4] (0x2)
7c935d73  6a22  push 0x22
7c935d75  6aff  push 0xff
7c935d77  e8b188fdff  call ntdll!ZwSetInformationProcess \(\leftarrow\) Invoke
7c935d7c  e9c076feff  jmp 7c91d441  \(\leftarrow\) NX is now disabled
...
7c91d441  5e  pop esi
7c91d442  c9  leave
7c91d443  c20400  ret 0x4

\(\leftarrow\) Return to jmp esp then jump into shellcode
Mitigation: Permanent flag

- Boot flag can force all applications to run with NX enabled (AlwaysOn)\(^{10}\)

- Processes can prevent future updates to execute flags
  - `NtSetInformationProcess`\(^{22}\) with flag 0x8

- Does not mitigate return into `VirtualProtect`
Recap: DEP (NX)

- Memory segments can be marked non-executable with hardware support
  - Stacks, heaps, etc

- Ret2libc can run malicious code without using shellcode

- It can also be used to disable NX and run shellcode
  - VirtualProtect
  - NtSetInformationProcess
A common thread

What is common about all of the exploitation techniques discussed so far?
A common thread

• Each technique generally relies on address space knowledge
  – Address used for a return address
  – Address used for an SEH handler
  – Address used for a library routine (ret2libc)

• What if the address space was unpredictable?
Mitigation: ASLR

• **Address Space Layout Randomization (ASLR)**\(^{[12]}\)
  – Images must be compiled with `/dynamicbase`

• Randomizes memory locations
  – Addresses are no longer predictable
Exploit: Partial overwrite

- Only the high-order two bytes are randomized in image mappings.
- Low-order two bytes can be overwritten to return into another location within a mapping:
  - Overwriting \texttt{0x1446047c} with \texttt{0x14461846}
- Target address can be used to pivot.
Exploit: non-reloc executables

- Not all binaries are compiled with relocation information
  - Executables often don’t have relocations (/fixed:yes)

- ASLR is only effective if all regions are randomized
Exploit: Brute force

- Vista ASLR randomizes most DLLs once per-boot

- Brute forcing addresses may be possible
  - No “forking” daemons in Windows
  - Vista service restart policy limits this

- Not as effective against Windows ASLR in most cases
Exploit: Information disclosure

- Application bugs may leak address space information
- Can be used to construct reliable return addresses
- Knowledge of image file version is all that is needed
Recap: ASLR

• Address space becomes unpredictable

• Exploits cannot assume the location of opcodes and other values

• Still, it has its weaknesses
  – Partial overwrite
  – Brute force
  – Information disclosure
Exploit techniques & mitigations

THE CHRONOLOGICAL EVOLUTION
Chronology on Windows

• **Attack**: Smashing the stack (Aug, 1996)

• **Mitigation**: Visual Studio 2002 (Feb, 2002)
  – First release of /GS[7]

• **Attack**: Overwrite variables[2] (Feb, 2002)

• **Attack**: SEH Overwrite[1] (Sep, 2003)
Chronology on Windows

• **Mitigation**: Visual Studio 2003 (Nov, 2003)
  – Arrays placed adjacent to GS cookie
  – /SAFESEH added [9]

• XP SP2 released (Aug, 2004)
  – Windows compiled with /GS and /SAFESEH
  – DEP

• **Attack**: Bypass NX [3] (Sep, 2005)
Chronology on Windows

- **Mitigation**: Visual Studio 2005 (Nov, 2005)
  - Arguments copied to safe locals for /GS

- **Mitigation**: ASLR (Nov, 2006)
  - Included with Windows Vista
  - Attacks against ASLR already existed

- **Attack**: Weak GS entropy \[5\] (May, 2007)
Exploit techniques & mitigations

WRAP UP
Looking toward the future

- Vista has formidable mitigations
  - GS, SafeSEH, Heap cookies, DEP, ASLR

- Easily exploitable issues have been found
  - Alexander Sotirov’s write-up on ANI

- Third parties have been slow to adopt

- Unlikely Vista will have a wormable flaw
Questions?
References


