Defenses against Code Injection Attacks

Angelos Stavrou, George Mason University
Code-Injection Attacks

Three essential stages

a. Inject attack code
b. Hijack control flow
c. Execute attacker code
Code Injection Defense

- **Strategies**
  - Detect and remove vulnerabilities (best)
  - Prevent code injection
  - Detect code injection
  - Prevent control flow hijacking
  - Prevent code execution

- **Stages of intervention**
  - Analyzing and compiling code
  - Linking objects into executable
  - Loading executable into memory
  - Running executable
Existing Approaches

- Security Extensions
  - Non-Executable Stack
  - MemGuard, StackGuard, ...
  - Libsafe

- NX Protection
  - Hardware vs. Software

- Randomization
  - Address Space Layout Randomization
  - Instruction Set Randomization
Non-Executable Stack

- Basic stack exploit can be prevented by marking stack segment as non-executable or randomizing stack location.
  - Code patches exist for Linux and Solaris.

- Problems:
  - Does not block more general overflow exploits:
    - Overflow on heap: overflow buffer next to func pointer.
  - Some apps need executable stack (e.g. LISP interpreters).
## Non-Executable Stack: Properties

<table>
<thead>
<tr>
<th></th>
<th>Attack Code Injection</th>
<th>Control Flow Hijacking</th>
<th>Attack Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Non-Executable Stack</strong></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
Security Extensions: MemGuard

- Main idea: prevent return address from changing via mark it as read-only.
- Extend VM model to protect user pages (such as return address in stack).
- Implementation details
  - GCC’s function_prologue and function_epilogue
- Flaw: performance penalties
  - Loading VM hardware is a privileged operation, and so the application process must trap to kernel mode to protect a word.
## Security Extensions: MemGuard (Properties)

<table>
<thead>
<tr>
<th></th>
<th>Code Injection Essential Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Attack Code injection</td>
</tr>
<tr>
<td>MemGuard</td>
<td>X</td>
</tr>
</tbody>
</table>
Main idea: the technique used to smash the stack currently always involve sequential memory writing.

- If the return address in stack was destroyed, the content before the return address must be destroyed, too.

- Keep a “canary word” before return address and check this word before function returns

Simple Demo
- http://nsfsecurity.pr.erau.edu/bom/StackGuard.html
move canary-word into register
push register

Function prologue: laying down a canary

move canary-word into register
exclusive-or register with top-of-stack
jump-if-not-zero to constant address .death-handler
add 4 to stack pointer
<normal return instructions here>
.death-handler

Function epilogue: checking a canary

Process stack while calling a function

- Local variable
- Canary word
- Return address

top of stack
bottom of stack
Flaw in static canary

- If the attacker can easily guess the canary value, then the attack string can include the canary word in the correct place; therefore, the canary checking routine in function epilogue may become useless.

  - Workaround?
    - randomize canary-word
Randomized Canary

Function prologue of randomized canary generator

move canary-index-constant into register
push canary-vector[register]
move canary-index-constant into register
move canary-vector[register] into register
exclusive-or register with top-of-stack
jump-if-not-zero to constant address .death-handler
add 4 to stack pointer
<normal return instructions here>
.death-handler

Function epilogue of randomized canary checker
<table>
<thead>
<tr>
<th>Code Injection Essential Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack Code injection</td>
</tr>
<tr>
<td>Control Flow Hijacking</td>
</tr>
<tr>
<td>Attack Code Execution</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StackGuard</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Libsafe (Avaya Labs)

- Dynamically loaded library (*LD_PRELOAD*)
- Intercepts calls to *strcpy* (dest, src)
  - Validates sufficient space in current stack frame: \(|\text{frame-pointer} - \text{dest}| > \text{strlen(src)}\)
  - If so, does *strcpy*.
  - Otherwise, terminates application.
### Security Extension: Libsafe (Properties)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Libsafe</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are those unsafe functions

- strcpy()
- strcat()
- strtok()
- sprintf()
- vsprintf()
- ...

NX Protection (Hardware)

Page Table

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Offset</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>X</td>
<td>RW</td>
</tr>
<tr>
<td>000001</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td>000010</td>
<td>100101</td>
<td>RW</td>
</tr>
<tr>
<td>000011</td>
<td>X</td>
<td>RW</td>
</tr>
<tr>
<td>000100</td>
<td>110110</td>
<td>R</td>
</tr>
<tr>
<td>000101</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td>000110</td>
<td>X</td>
<td>RW</td>
</tr>
<tr>
<td>000111</td>
<td>X</td>
<td>RW</td>
</tr>
<tr>
<td>001000</td>
<td>001011</td>
<td>RW</td>
</tr>
</tbody>
</table>

Physical Memory

Question: Can we implement a software-based NX protection?
NX Protection (Properties)

<table>
<thead>
<tr>
<th></th>
<th>Attack Code injection</th>
<th>Control Flow Hijacking</th>
<th>Attack Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
Discussion on NX

- Does not prevent stack overflow
- Prevents executing injected code
- Exploit can still call system functions with arbitrary arguments
  - e.g. system("/bin/sh")
  - Technique is called return-into-libc
Return-to-libc Attack

- Overflows a buffer on stack
  - e.g. local array variable
- Overwrites the return address with that of a library function
  - e.g. `system(3)`
- Also writes on stack
  - False return address for library function (**RET2**)
  - Function arguments

```
Arg
RET2
RET
EBP
```

```
/bin/sh
```

```
buf[0..3]
```
<table>
<thead>
<tr>
<th></th>
<th>Attack Code Injection</th>
<th>Control Flow Hijacking</th>
<th>Attack Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Executable Stack</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>MemGuard</td>
<td>X</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>StackGuard</td>
<td>✓</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>LibSafe</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>NX</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>
Address Space Layout Randomization

- ASLR renders exploits *which depend on predetermined memory addresses* useless by randomizing the layout of the virtual memory address space.
  - Base addresses of stack, heap, and code segment
- Randomization can be done at compile- or link-time, or by rewriting existing binaries
- Several implementations available
  - E.g., PaX ASLR
Example Shellcode from Windows Slammer worms

- **STRING BUFFER**: 375 bytes
- **STRING 1**: buff[128]
- **LOCAL VARIABLES**
- **SAVED FRAME POINTER**
- **RETURN ADDRESS**: ARG0, ARG1, ARG2
- **WORM CODE**

**SQLSORT.DLL**

- **42B0C9DC:** JMP ESP
- **LoadLibraryA()**
- **GetProcAddress()**
Is ASLR Enough?

- Two possible attacks
  - De-Randomization Attacks
    - Repetitively guess randomized address
  - Code Spraying Attacks
    - Spraying injected attack code
De-Randomization Attacks

Stack Frame

- Code
- Buf
- Ret
- Pad

Step 1

3 GB
De-Randomization Attacks

Step 2

Stack Frame

• Code
• Buffer
• Retad
• Crash

Pad

3 GB
De-Randomization Attacks

216 seconds (avg.) to de-randomize! [Shacham+, CCS’04]
Code Spraying Attacks

- Exploit a buggy application and “spray” attack code repetitively throughout large write-able user-level memory areas
  - e.g., 256MB, which leaves only 4 bit entropy in the current 32 bit architecture for attackers to guess the location of a attack code replica
A Real-world Example (IE JView Profiler Vul.)

1. Preparing a basic block containing the NOP-sled and shellcode
2. Replicating the basic block into 750 other blocks, each of which contains the NOP-sled and shellcode.

In total, the shellcode is sprayed into 750 * 256K or **187.5M** bytes.

3. Triggering the JView Profiler vulnerability (MS05-037)

4G/2/187.5M = 10.9 < 16 \(\Rightarrow\) **Less than 4 entropy bits of randomization!**
Example Shellcode from Windows Slammer worms

SQLSORT.DLL

42B0C9DC: JMP ESP

Import
Address
Directory

LoadLibraryA()
GetProcAddress()

Example Shellcode from Windows Slammer worms

Top of stack
String 1
buff[128]
LOCAL VARIABLES
SAVED FRAME POINTER
RETURN ADDRESS
ARG0
ARG1
ARG2
WORM CODE
Randomizing library API interfaces

- Two essential phrases
  - Load-time name randomization
    - Library API renaming: sendto ➔ 8OmsZ3wf
  - Run-time name de-randomization
    - Library API reverse renaming: 8OmsZ3wf ➔ sendto

- For normal programs
  - sendto ➔ 8OmsZ3wf ➔ sendto

- For injected shellcode
  - sendto ➔ sendto ➔ YBHdgv2Q$
MSBlast Worm Shellcode

RanSys blocks MSBlast worms here
Instruction Set Randomization (ISR)

- ISR renders exploits which depend on the predetermined instruction set useless by randomizing current instruction set and making it externally unpredictable.
- Randomization is usually done at execution-time:
  - Each program has a different and secret instruction set.
  - Use translator to randomize instructions at load-time.
- Two research prototypes:
  - RISE [Barrante+, CCS’03] and ISR x86 [Kc+, CCS’03]
RISE: Loading Binary

ELF binary file

Code

Data

Valgrind / RISE

Key

Memory

Scrambled Code

Data
RISE: Executing Code

Valgrind / RISE

Key

Hardware

Code

Memory

Scrambled Code

Data
RISE: Foreign Code

Injected from network
Code

Valgrind / RISE
Key
+

SIGILL

Hardware

Memory
Scrambled Code
Data
Code
x86 shellcode – de-randomized

1. **Prepare parameters**
   - `0xeb 1f`: jmp IP + 1f
   - `0x5e`: pop %esi
   - `0x89 76 08`: mov %esi, 08(%esi)
   - `0x31 c0`: xor %eax, %eax
   - `0x88 46 07`: mov %al, 07(%esi)
   - `0x89 46 0c`: mov %eax, 0c(%esi)
   - `0xb0 0b`: mov 0b, %al
   - `0x89 f3`: mov %esi, %ebx
   - `0x8d 4e 08`: lea 08(%esi), %ecx
   - `0x8d 56 0c`: lea 0c(%esi), %edx
   - `0xcd 80`: int $0x80
   - `0x31 db`: xor %ebx, %ebx
   - `0x89 d8`: mov %ebx, %eax
   - `0x40`: inc %eax
   - `0xcd 80`: int $0x80
   - `0xe8 dcfffffff`: call 0xdfffffff

2. **Execve**

   ```
   0xcd d6 : int $0xd7
   0x24 c9 : and $0xc9,%al
   0x24 97 : and $0x97,%al
   0xad : lods %ds:(%esi),%eax
   0xbf 2c f8 e4 41 : mov $0x41e4f82c,%edi
   0x62 ce : bound %ecx,%esi
   0xad : lods %ds:(%esi),%eax
   0x8f 28 : popl (%eax)
   0x79 2f : jns 4a <foo+0x4a>
   0x40 : inc %eax
   0xd7 : xlat %ds:(%ebx)
   0x44 : inc %esp
   0x6a c1 : push $0xffffffff
   0xa9 9f 28 04 a4 : test $0xa404289f,%eax
   0xf8 : clc
   0xff 40 fc : incl 0xffffffff(%eax)
   0x89 e9 : mov %ebp,%ecx
   0x49 : dec %ecx
   0xcc : int3
   0x1e : push %ds
   0xdb 36 : (bad) (%esi)
   0xdb 59 b4 : fistpl 0xffffffffb4(%ecx)
   ```

Highly likely to crash
Discussion

- Any limitation in ISR?
  - Performance!!!
- Can we improve it?
Randomizing system call numbers

- Two essential phrases
  - Load-time randomization
    - System call remapping: eax → eax'
  - Run-time de-randomization
    - System call reverse remapping: eax' → eax

- For normal programs
  - eax → eax' → eax

- For injected shellcode
  - eax → eax → eax'?
Example Shellcode from Linux Lion Worms

```c
char shellcode[] =
"… …"
"eb 14"    /* jmp <shellcode+0x68> */
"31 c0"    /* xorl %eax,%eax */
"5b"       /* popl %ebx */
"8d 4b 14" /* leal 0x14(%ebx),%ecx */
"89 19"    /* movl %ebx,(%ecx) */
"89 43 18" /* movl %eax,0x18(%ebx) */
"88 43 07" /* movl $0xb,%al */
"31 d2"    /* xorl %edx,%edx */
"b0 0b"    /* movb $0xb,%al */
"cd 80"    /* int $0x80 */
"e8 e7 ff ff ff ff" /* call <shellcode+0x54> */
"2f 62 69 6e 3f 73 68" ; "/bin/sh"
"90 90 90 90 90 90 90 90"
```

**sys_execve** → **/bin/sh**

**sys_fchown**

---

**eax** 0xb

**ebx**

**ecx**

**edx** \0

---

**shellcode+0x54**

**shellcode+0x68**

---

without RandSys

with RandSys
## A Comparison

<table>
<thead>
<tr>
<th></th>
<th>Code Injection Essential Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Attack Code injection</td>
</tr>
<tr>
<td>ASLR</td>
<td>×</td>
</tr>
<tr>
<td>ISR</td>
<td>×</td>
</tr>
</tbody>
</table>