CSC501
Operating Systems Principles

Code Injection Defense
Outline

- Code-Injection Recap
- Code-Injection Defense
Code-Injection Attacks

1. Inject attack code
2. Hijack control flow
3. Execute attacker code
Code Injection Defense

q Strategies
  Q Detect and remove vulnerabilities (best)
  Q Detect and prevent code injection
  Q Detect and prevent control flow hijacking
  Q Detect and prevent code execution

q Stages of intervention
  Q Analyzing and compiling code
  Q Linking objects into executable
  Q Loading executable into memory
  Q 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).
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Executable Stack</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
</tr>
</tbody>
</table>
Security Extensions -- MemGuard

q Main idea: prevent return address from changing via mark it as read-only.

q Extend VM model to protect user pages (such as return address in stack).

q Implementation details
  Q GCC’s function_prologue and function_epilogue

q Flaw: *performance penalties*
  Q Loading VM hardware is a privileged operation, and so the application process must trap to kernel mode to protect a word.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MemGuard</td>
<td>✗</td>
<td>✓</td>
<td>—</td>
</tr>
</tbody>
</table>
Security Extensions -- StackGuard

q Main idea: the technique used to smash the stack currently always involve sequential memory writing.
  Q If the return address in stack was destroyed, the content before the return address must be destroyed, too.
q Keep a "canary word" before return address and check this word before function returns
q Simple Demo
  Q http://nsfsecurity.pr.erau.edu/bom/StackGuard.html
The Canary

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
move canary-word into register
push register
Process stack while calling a function
Local variable
Canary word
Return address
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.

Q  Workaround?

   - randomize canary-word
Randomized Canary

move canary-index-constant into register
push canary-vector[register]

Function prologue of randomized canary generator

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>Attack Code Injection</th>
<th>Control Flow Hijacking</th>
<th>Attack Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Security Extension -- Libsafe

q Libsafe (Avaya Labs)
Q Dynamically loaded library (*LD_PRELOAD*)
Q Intercepts calls to *strcpy* (dest, src)
   ÿ Validates sufficient space in current stack frame:
      |frame-pointer – dest| > strlen(src)
   ÿ If so, does strcpy.
      Otherwise, terminates application.
<table>
<thead>
<tr>
<th>Libsafe</th>
<th>Attack Code Injection</th>
<th>Control Flow Hijacking</th>
<th>Attack Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
What are those unsafe functions

- `strcpy()`
- `strcat()`
- `strtok()`
- `sprintf()`
- `v sprintf()`
- ...
Question: Can we implement a software-based NX protection?
Software-Based NX Protection

1. Logical Memory
2. Instruction Fetch
3. D-TLB
4. Page Table
5. O.S.
6. Physical Memory
7. Update PT
8. Restart Process

TLB miss
Get page from backing store

Bring in page
Software-Based NX Protection
Software-Based NX Protection

1. Logical Memory

2. Instruction Fetch

3. Data Fetch

4. Page fault

5. Get page from backing store

6. Bring in page

7. Update PTE

8. Restart Process

Non-eXecute
Software-Based NX Protection

Questions:

Q  Can we have two page tables for one process?
Q  How about CR3 – page directory base register?

See work-around by PaX

Q  Exploiting segment support (SEGMEXEC)
Q  Overloading “supervisor bit” (PAGEEEXEC)

http://www.answers.com/topic/pax-4
<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>NX</strong></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
Discussion on NX

- Prevents executing injected code

- Limitation:
  - 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
buf[0..3]
"/bin/sh"
```
<table>
<thead>
<tr>
<th>Non-Executable Stack</th>
<th>MemGuard</th>
<th>StackGuard</th>
<th>LibSafe</th>
<th>NX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack Code injection</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Control Flow Hijacking</td>
<td>X</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Attack Code Execution</td>
<td>✓</td>
<td>—</td>
<td>—</td>
<td>✓</td>
</tr>
</tbody>
</table>
Address Space Layout Randomization (ASLR)

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

When to do randomization:

Compile- or link-time, or by rewriting existing binaries
Example Shellcode from Windows Slammer worms

SQLSORT.DLL

42B0C9DC: JMP ESP

JMP ESP

Import Address Directory

LoadLibraryA()

GetProcAddress()

GetProcAddress()

WORM CODE

String 1
buff[128]

LOCAL VARIABLES

ARG0

ARG1

ARG2

RETURN ADDRESS

Top of stack

string buffer

ESP = Stack Pointer

Register

375 bytes

ASLR
## 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>
</tbody>
</table>
Next Lecture

q Protection – File System