BUFFER OVERFLOW ATTACKS AND DEFENSES

EDWARD CHOW
Outline of the Talk

- History of Buffer Overflow Attacks
- Buffer Overflow Attack and related Background Knowledge
  - Linux VirtualMemory Map
- Shellcode
- Egg: No-ops/shellcode/returnAddresses
- Countermeasures:
  - StackGuard
  - StackShield
  - W⊕XPage
  - Address Space Randomization
- Hardware Assisted Non-executable page protection:
  - AMD NX bit
  - Intel Execute Disable Bit
Related Literature

“Smashing The Stack For Fun And Profit,” by Aleph One

“On the Effectiveness of Address-Space Randomization,” by Shacham et al at Stanford's applied crypto group

The material presented here are adapted from

bufferOverflow/cs155stanford/04-buf-overflow.ppt

http://www.cs.utexas.edu/~shmat/courses/cs378_spring05/16overflow.ppt
Buffer Overflow Attacks

- Extremely common bug.
  - First major exploit: 1988 Internet Worm Robert Morris. fingerd.

- 15 years later: ≈ 50% of all CERT advisories:
  - 1998: 9 out of 13
  - 2001: 14 out of 37
  - 2003: 13 out of 28

- Often leads to total compromise of host.

- Steps in developing buffer overflow attacks:
  - Locate buffer overflow within an application.
  - Design an exploit.
We’ll look at the Morris worm in more detail when talking about worms and viruses.

One of the worm’s propagation techniques was a buffer overflow attack against a vulnerable version of fingerd on VAX systems.

By sending special string to finger daemon, worm caused it to execute code creating a new worm copy.

Unable to determine remote OS version, worm also attacked fingerd on Suns running BSD, causing them to crash (instead of spawning a new copy).
Worm was released in 1988 by Robert Morris
- Graduate student at Cornell, son of NSA chief scientist
- Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
- Now a computer science professor at MIT

Worm was intended to propagate slowly and harmlessly measure the size of the Internet
Due to a coding error, it created new copies as fast as it could and overloaded infected machines
$10-100M worth of damage
Buffer: A contiguous block of computer memory, can be used for
- Data: variables (static/global, dynamic/local), arrays
- Code: user programs, shared libraries, kernel programs.

To shield User/kernel programs from each other, virtual memory is used.

Within a virtual memory address space, different OS'/CPUs have different ways to allocate buffers.

On Linux, static/global variables allocated at load time on the data segment, dynamic/local variables are allocated at run time on the stack.
Virtual Memory Address and Page Memory Systems

- User Process 1
  - Data Pages
  - Code Pages

- User Process 2
  - Data Pages
  - Code Pages

- User Process n
  - Data Pages
  - Code Pages

• Virtual Memory Address

CPU -> VM_Address -> MMU -> PM_Address

- Protection mode
- Relative Page #
- User process #

Page(1,9) R
Page(1,8) RW
Page(1,7) RW
Page(1,1) E
Page(2,9) R
Page(2,7) RW
Page(3,9) R
Page(3,7) RW
Page(3,6) RW
Page(3,1) E
Page(2,6) RW
Page(2,5) RW
Page(2,2) E
Page(2,1) E

MMU: Memory Management Unit

Physical Memory

Address Mapping
Page loading

Data/Code
### The Linux Virtual Memory Map (as seen by a UserSpace program)

<table>
<thead>
<tr>
<th>Starts at</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffffffff</td>
<td>End of the universe</td>
</tr>
<tr>
<td>fffe000</td>
<td>vsyscall table (new in 2.5.x)</td>
</tr>
<tr>
<td>c0000000</td>
<td>Off limits, reserved for the kernel</td>
</tr>
<tr>
<td>bfffffff</td>
<td>Process stack (grows down)</td>
</tr>
<tr>
<td>bffff000</td>
<td>Process heap (grows up)</td>
</tr>
<tr>
<td>40000000</td>
<td>Libraries</td>
</tr>
<tr>
<td>zzzzzzzzzz</td>
<td>Unused</td>
</tr>
<tr>
<td>yyyyyyyyy</td>
<td><code>.bss</code>, uninitialised program data</td>
</tr>
<tr>
<td>xxxxxxxxx</td>
<td><code>.data</code> segment, initialised program data</td>
</tr>
<tr>
<td>08048000</td>
<td><code>.text</code> segment, program code</td>
</tr>
<tr>
<td>00000000</td>
<td>Unmapped to trap <code>NULL</code> pointers</td>
</tr>
</tbody>
</table>
Memory Layout of Linux Process

- **Kernel Memory**
  - 0xC0000000
- **User Stack**
  - 0x40000000
- **Shared libraries, mmap, shared mem**
  - 0x40000000
- **Run time heap**
  - 0x08048500
- **Text Segment, BSS**
  - 0x08048500
- **Unused**
  - 0x08048500

- **Stack Pointer**
  - %esp
- **Brk()**
  - Loaded from exec
char buf[32];
struct Node* bn;
int main(int argc, char* argv[]) {
    return function(argv[1]);
}

int function(char* str1) {
    char a[32];
    printf("Hello World!");
    gets(a);
    bn = malloc(sizeof(struct Node));
    *(buf-32)=0xbfffffeb4;
    strcpy(buf, str1);
    return lookup(a, bn);
}

Where are Buffers Allocated & Buffer Overflow Opportunities?

Kernel Memory
User Stack
Shared libraries, mmap, shared mem
Run time heap
Text Segment, BSS
Unused

0xC0000000
0x40000000
0x08048500
0

1/31/2007
CS591 / C. Edward Chow
Page 11
Stack Frame

- Parameters
- Return address
- Stack Frame Pointer
- Local variables

SP → Stack Growth
Suppose a web server contains a function:

```c
char a[30];
void func(char *str) {
    char buf[128];
    strcpy(buf, str); strcpy(a, str);
    do-something(buf);
}
```

When the function is invoked the stack looks like:

<table>
<thead>
<tr>
<th>Lower memory address</th>
<th>buf</th>
<th>sfp</th>
<th>ret-addr</th>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>top of stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What if `*str` is 136 bytes long? After `strcpy`:

<table>
<thead>
<tr>
<th>Direction of copy operation</th>
<th>*str</th>
<th>sfp*</th>
<th>Ret*</th>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buf+132</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>top of stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Main problem: no range checking in `strcpy()`.

Suppose `*str` is such that after `strcpy` stack looks like:

```
*str  ret  Code for P
```

Program P: `exec( "/bin/sh" )`

(exact shell code by Aleph One)

When `func()` exits, the user will be given a shell `!!`

Note: attack code runs `in stack`.

To determine `ret` guess position of stack when `func()` is called.
What is Needed to Create Buffer Overflow Exploit?

- Understanding C functions and the stack.
- Some familiarity with machine code.
- Know how systems calls are made.
- The exec() system call.

- Attacker needs to know which CPU and OS are running on the target machine.

  - Our examples are for x86 running Linux.
  - Details vary slightly between CPU’s and OS:
    - Stack growth direction.
    - big endian vs. little endian
Main problem:
- `strcpy()`, `strcat()`, `sprintf()` have no range checking.
- "Safe" versions `strncpy()`, `strncat()` are misleading
  - `strncpy()` may leave buffer unterminated.
  - `strncpy()`, `strncat()` encourage off by 1 bugs.

Defenses:
- Type safe languages (Java, ML). Legacy code?
- Mark stack as non-execute. Random stack location.
- Static source code analysis.
- Run time checking: StackGuard, Libsafe, SafeC, (Purify).
- Many more … (covered later in course)
gcc version 4.0.0 20050519 (Red Hat 4.0.0-8) on Athena (FC4) by default mark the executable does not require executable stack. This is a buffer overflow prevention measure. Therefore when running the testsc example in Aleph's paper on Fedora Core 4, you will get segmentation fault. You change it by using "execstack -s" to mark testsc as requiring executable stack, then we can see the buffer overflow effect. On fc4.csnet, testc flag is not set. You need to recompile testsc to have it set with the protection.

[chow@athena src]$ execstack -q testsc
- testsc
[chow@athena src]$ ./testsc
Segmentation fault
[chow@athena src]$ execstack -s testsc
[chow@athena src]$ ./testsc
sh-3.00$ exit
exit
[chow@athena src]$ 

See man page of execstack at
http://www.pk.edu.pl/cgi-bin/man-cgi?execstack
Basic stack exploit can be prevented by marking stack segment as non-executable.

- NX Bit on AMD Athlon 64, Intel P4 “Prescott”.
  - NX bit in every Page Table Entry (PTE)
  - Support in SP2. Code patches exist for Linux, Solaris.

Problems:
- Does not defend against `return-to-libc’ exploit.
  - Overflow sets ret-addr to address of libc function.
- Some apps need executable stack (e.g. LISP interpreters).
- Does not block more general overflow exploits:
  - Overflow on heap: overflow buffer next to func pointer.
Statically check source to detect buffer overflows.
Several consulting companies.

Can we automate the review process?
Several tools exist:
Coverity (Engler et al.): Test trust inconsistency.
Microsoft program analysis group:
PREfix: looks for fixed set of bugs (e.g. null ptr ref)
PREfast: local analysis to find idioms for prog errors.

Find lots of bugs, but not all.
Run time checking: StackGuard

Many many run-time checking techniques …

Here, only discuss methods relevant to overflow protection.

Solutions 1: StackGuard (WireX)

Run time tests for stack integrity.

Embed “canaries” in stack frames and verify their integrity prior to function return.

“StackGuard works by inserting a `guard" value (called a `canary", \textit{as in how this bird was used in mines}) in front of the return address; “

\begin{tabular}{c c c c c c}
\text{Frame 1} & & & & & \\
local & canary & sfp & ret & str & \\
\text{Frame 2} & & & & & \\
local & canary & sfp & ret & str & \\
\end{tabular}
Canary Types

- **Random canary:**
  - Choose random string at program startup.
  - Insert canary string into every stack frame.
  - Verify canary before returning from function.
  - To corrupt random canary, attacker must learn current random string.

- **Terminator canary:**
  - Canary = 0, newline, linefeed, EOF
  - String functions will not copy beyond terminator.
  - Hence, attacker cannot use string functions to corrupt stack. Why? I thought attacker has control on the data.
StackGuard (Cont.)

- StackGuard implemented as a GCC patch.
  - Program must be recompiled.

- Minimal performance effects: 8% for Apache.

- Newer version: PointGuard.
  - Protects function pointers and setjmp buffers by placing canaries next to them.
  - More noticeable performance effects.

- Note: Canaries don’t offer fullproof protection.
  - Some stack smashing attacks can leave canaries untouched.
ProPolice (IBM) New Name: Stack-Smashing Protector -
GCC 3.4.1, GCC 4.1 stage 2.

- Rearrange stack layout to prevent ptr overflow.
- Random local variables allocation

Stack Guard variants - ProPolice

String Growth
- args
- ret addr
- SFP
- CANARY
- arrays
- Local variables

No arrays or pointers

Ptrs, but no arrays
Non executable stack.

**Compiler /GS option:**
- Combination of ProPolice and Random canary.
- Triggers UnHandledException in case of Canary mismatch to shutdown process.

**Litchfield vulnerability report.**
- Overflow overwrites exception handler.
- Redirects exception to attack code.
Solutions 2: Libsafe (Avaya Labs)

- Dynamically loaded library.
- Intercepts calls to `strcpy(dest, src)`
  - Validates sufficient space in current stack frame:
    - `|frame-pointer – dest| > strlen(src)`
  - If so, does `strcpy`. Otherwise, terminates application.
More methods ...

- **StackShield**
  - At function prologue, copy return address RET and SFP to “safe” location (beginning of data segment)
  - Upon return, check that RET and SFP is equal to copy.
  - Implemented as assembler file processor (GCC)

- **Randomization:**
  - **PaX ASLR:** Randomize location of libc.
    - Attacker cannot jump directly to exec function.
  - **Instruction Set Randomization (ISR)**
    - Attacker cannot execute its own code.