Buffer Overflow

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
x86-64/Linux Memory Layout

• Stack
  – Runtime stack (8MB limit)

• Heap
  – Dynamically allocated as needed
  – When call malloc(), calloc(), new()

• Data
  – Statically allocated data
  – e.g. global vars, static vars, string constants

• Text / Shared libraries
  – Executable machine instructions
  – Read-only
# x86-64 Addresses Example

```c
#include <stdio.h>
#include <stdlib.h>

int g = 1;
int main(void) {
    char *p = malloc(100);
    printf("main() = %p\n", main);
    printf("&g = %p\n", &g);
    printf("&p = %p\n", &p);
    printf("p = %p\n", p);
}

$ gcc -Og -g mem.c
$ ./a.out
main() = 0x4005f6
&g = 0x601048
&p = 0x7fff07b94b70
p = 0x1ece010
```

The diagram illustrates the memory layout with the following sections:

- **Stack**: Top of the diagram, represented by a light blue section.
- **Shared Libraries**: Above the heap, represented by a light green section.
- **Heap**: Below the shared libraries, represented by a green section.
- **Data**: Below the heap, represented by a red section.
- **Text**: Below the data, represented by a yellow section.

The hexadecimal addresses shown in the diagram correspond to the memory layout:

- **Text**: 0x400000
- **Data**: 0x7FFFFFFFFFFF
- **Stack**: 0x7FFFFFFF
- **Heap**: 0x7fff07b94b70
- **Shared Libraries**: Above the heap.
- **Local Variables**: Addresses for `g`, `main`, and `p`.

The `malloc` function allocates memory, and the `printf` statements output the addresses of `main`, `g`, and `p`.
Vulnerable Buffer Code

/* Echo Line */
void echo()
{
    // Way too small!
    char buf[4];
    gets(buf);
    puts(buf);
}

int main()
{
    printf("Type: ");
    echo();
    return 0;
}
String Library Code

• Implementation of Unix function gets()
  – No way to specify limit on number of characters to read

```c
char *gets(char *dest) {   // Get string from stdin
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

– Similar problems with other Unix functions
  • strcpy: copies string of arbitrary length
  • scanf / fscanf / sscanf, given %s conversion specification
Buffer Overflow Disassembly

- **echo()**:  

```
00000000004006cf <echo>:
4006cf: 48 83 ec 18  sub $0x18,%rsp
4006d3: 48 89 e7   mov %rsp,%rdi
4006d6: e8 a5 ff ff ff callq 400680 <gets>
4006db: 48 89 e7   mov %rsp,%rdi
4006de: e8 3d fe ff ff callq 400520 <puts@plt>
4006e3: 48 83 c4 18 add $0x18,%rsp
4006e7: c3 retq
```

- **main()**:  

```
...  
4006ec: b8 00 00 00 00 mov $0x0,%eax
4006f1: e8 d9 ff ff ff callq 4006cf <echo>
4006f6: 48 83 c4 08 add $0x8,%rsp
...  
```
Buffer Overflow (1)

- Before call to `gets()`

Stack Frame for `main`

Return Address (8 bytes)

20 bytes unused

buf

%rsp

echo:

- `4006cf: sub $0x18,%rsp`
- `4006d3: mov %rsp,%rdi`
- `4006d6: callq 400680 <gets>`
- `4006db: mov %rsp,%rdi`
- `4006de: callq 400520 <puts@plt>`
- `4006e3: add $0x18,%rsp`
- `4006e7: retq`

main:

...`

- `4006ec: mov $0x0,%eax`
- `4006f1: callq 4006cf <echo>`
- `4006f6: add $0x8,%rsp`

...`
Buffer Overflow (2)

• Before call to gets()

```assembly
main:
    ...  
4006ec: mov $0x0,%eax  
4006f1: callq 4006cf <echo>  
4006f6: add $0x8,%rsp  
    ...  
```

```assembly
echo:
    4006cf: sub $0x18,%rsp  
4006d3: mov %rsp,%rdi  
4006d6: callq 400680 <gets>  
4006db: mov %rsp,%rdi  
4006de: callq 400520 <puts@plt>  
4006e3: add $0x18,%rsp  
4006e7: retq  
```

Stack Frame for main

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>f6</td>
</tr>
</tbody>
</table>

20 bytes unused

buf ← %rsp

%rip →
Buffer Overflow (3)

- Overflowed buffer, but did not corrupt state

```
buf
$ ./bufdemo
Type: 01234567890123456789012
01234567890123456789012
```
Buffer Overflow (4)

- Overflowed buffer, and corrupted return pointer

```
Stack Frame for main

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

buf ← %rsp
```

```
echo:
4006cf:  sub   $0x18,%rsp
4006d3:  mov   %rsp,%rdi
4006d6:  callq 400680 <gets>
4006db:  mov   %rsp,%rdi
4006de:  callq 400520 <puts@plt>
4006e3:  add   $0x18,%rsp
4006e7:  retq
```

```
$ ./bufdemo
Type: 012345678901234567890123
Segmentation fault (core dumped)
```
Buffer Overflow Attack

• Malicious use of buffer overflow
  – Input string contains byte representation of executable code
  – Overwrite return address A with address of buffer B
  – When P() executes `ret`, will jump to exploit code

```c
void P() {
    Q();
    ...
}

void Q() {
    char buf[64];
    gets(buf);
    ...
}
```

Stack after call to `gets()`
Exploits Using Buffer Overflows

• Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines

• Distressingly common in real programs
  – Programmers keep making the same mistakes 😞
  – Recent measures make these attacks much more difficult

Google Security Blog
The latest news and insights from Google on security and safety on the Internet

CVE-2015-7547: glibc getaddrinfo stack-based buffer overflow
February 16, 2016
Internet Worm (1988)

• Exploited a few vulnerabilities to spread
  – Early versions of the finger server (fingerd) used gets() to read the argument sent by the client:
    • finger kildong@skku.edu
  – Worm attacked fingerd server by sending phony argument:
    • finger “exploit-code padding new-return-address”
    • exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker

• Once on a machine, scanned other machines to attack
  – Invaded ~6000 computers in hours (10% of the Internet)
  – The young author of the worm was prosecuted
  – CERT(Computer Emergency Response Team) formed @ CMU
Worms vs. Viruses

• Worm: A program that
  – Can run by itself
  – Can propagate a fully working version of itself to other computers

• Virus: Code that
  – Adds itself to other programs
  – Does not run independently

• Both are (usually) designed to spread among computers and to wreak havoc
Code Red Worm (2001)

• History
  – June 18, 2001, Microsoft announces buffer overflow vulnerability in IIS Internet Server
  – July 19, 2001, Over 250,000 machines infected by new virus in 9 hours
  – White house must change its IP address. Pentagon shut down public WWW servers for day

• Received strings of form

GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN....NNNNNNN NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u819 0%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0" 400 325 "-" "-"
Code Red Worm (2001)

- Code Red exploit code
  - Starts 100 threads running
  - Spread self
    - Generate random IP addresses & send attack string
    - Between 1st & 19th of month
  - Denial of service attack to www.whitehouse.gov
    - Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Between 21st & 27th of month
  - Deface server’s home page
    - After waiting 2 hours
Nimda Worm (2001)

- Five different infection methods
  - Via e-mail
  - Via open network shares
  - Via browsing of compromised web sites
  - Exploitation of various Microsoft IIS 4.0/5.0 directory traversal vulnerabilities
  - Via back doors left behind by the Code Red II and Sadmind/IIS worms

- One of the most widespread virus/worm
SQL Slammer Worm (2003)

• History
  – Exploited two buffer overflow bugs in Microsoft’s SQL Server and Desktop Engine
  – Infected 75,000 victims within 10 minutes
  – Generate random IP addresses and send itself out to those addresses, slowing down Internet traffic dramatically
  – Jan. 25 nationwide Internet shutdown in South Korea

30 minutes after release
Avoiding Buffer Overflow

• Use library routines that limit string lengths
  – fgets() instead of gets()
  – strncpy() instead of strcpy()
  – Don’t use scanf() with %s conversion specification
    • Use fgets() to read the string
    • Or use %ns where n is a suitable integer

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
System-Level Protections

• Randomized stack offsets
  – At start of program, allocate random amount of space on stack
  – Makes it difficult for hacker to predict beginning of inserted code

• Executable space protection
  – Mark certain areas of memory as non-executable (e.g. stack)
  – Requires hardware assistance: x86-64 added explicit “execute” permission
Stack Canaries

• Idea
  – Place special value (“canary”) on stack just beyond buffer
  – Check for corruption before exiting function

• GCC implementation
  – -fstack-protector (now the default)

```
$ ./bufdemo
Type: 01234567
01234567
$ ./bufdemo
Type: 012345678
012345678
*** stack smashing detected ***: ./bufdemo terminated
Aborted (core dumped)
```
Summary

• Memory layout
  – OS/machine dependent (including kernel version)
  – Basic partitioning: stack, data, text, heap, shared libraries found in most machines

• Avoiding buffer overflow vulnerability
  – Important to use library routines that limit string lengths

• Working with strange code
  – Important to analyze nonstandard cases
    • e.g. What happens when stack corrupted due to buffer overflow?
  – Helps to step through with GDB