Control Hijacking Attacks

Cătălin Hrițcu

May 15, 2009
Substituting Prof. Backes
Control hijacking attacks

- **Attacker’s goal:**
  - Take over target machine (e.g. web server)
  - Execute arbitrary code on target by hijacking application control flow
This lecture: **attacks!**

- **Buffer overflows**
  - Stack-based attacks (stack smashing)
  - Heap-based attacks
  - Return-to-libc and return-oriented programming
- **Integer overflow attacks**
- **Format string vulnerabilities**

- **Project 1: writing exploits**
Assumptions are vulnerabilities

• How to successfully attack a system:
  1) Discover what assumptions were made
  2) Craft an exploit outside those assumptions
  3) Profit

• Two assumptions often exploited:
  – Target buffer is large enough for source data
    • Buffer overflows deliberately break this assumption
  – Computer integers behave like math integers
    • Integer overflows violate this assumption
Assumptions about control flow

• We write our code in languages that offer several layers of abstraction over machine code; even C
  – High-level statements: “=” (assign), “;” (seq), if, while, for, etc.
  – Procedures / functions

• Naturally, our execution model assumes:
  – Basic statements (e.g. assign) are atomic
  – Only one of the branches of an if statement can be taken
  – Functions start at the beginning
  – They (typically) execute from beginning to end
  – And, when done, they return to their call site
  – Only the code in the program can be executed
  – The set of executable instructions is limited to those output during compilation of the program
Assumptions about control flow

• We write our code in languages that offer several layers of abstraction over machine code; even C
  – High-level statements: “=” (assign), “;” (seq), if, while, for, etc.
  – Procedures / functions

• But, actually, at the level of machine code
  – Each basic statement compiled down to many instructions
  – There is no restriction on the target of a jump
  – Can start executing in the middle of functions
  – A fragment of a function may be executed
  – Returns can go to any program instruction
  – Dead code (e.g. unused library functions) can be executed
  – On the x86, can start executing not only in the middle of functions, but in the middle of instructions!
BUFFER OVERFLOWS
Buffer overflows

- Extremely common bug
- First major exploit: 1988 Internet Worm (targeted fingerd)

≈20% of all vuln.

2005-2007: ≈ 10%

Source: NVD/CVE
Many unsafe C lib functions

strncpy (char *dest, const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf ( const char *format, … )
sprintf (char * str, const char * format, … )

• “Safe” versions sometimes misleading
  – strncpy() leaves buffer unterminated if strlen(src) ≥ length arg.
  – strncpy(), strncat() encourage off by 1 bugs
    (dest buffer needs to have at least strlen(src) + 1 bytes allocated)
Eliminating unsafe functions doesn’t fix everything

• It could break things even more though (legacy code)
• Vulnerable code often written using explicit loops and pointer arithmetic

Not only this is vulnerable:

```c
int is_file_foobar( char* one, char* two ) {
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}
```

But also this:

```c
int is_file_foobar_using_loops( char* one, char* two ) {
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    char* b = tmp;
    for( ; *one != '\0'; ++one, ++b ) *b = *one;
    for( ; *two != '\0'; ++two, ++b ) *b = *two;
    *b = '\0';
    return strcmp( tmp, "file://foobar" );
}
```
Finding buffer overflows: fuzzing

• To find overflow:
  – Run target app on local machine
  – Issue requests with long strings that end with “$$$$$”
  – If app crashes,
    search core dump for “$$$$$” to find overflow location

• Many automated tools exist: called fuzzers

• Then use disassemblers and debuggers to construct exploit
  – The GNU Project Debugger (GDB) – free software
  – IDA-Pro – commercial
Buffer overflows

STACK-BASED ATTACKS
What is needed for building exploits

- Understanding C functions and the stack
- Some familiarity with machine code
- Know how systems calls are made (e.g. exec)
  - For project you will use “off-the-shelf” payload: “\texttt{shellcode}”

- Attacker needs to know which CPU and OS are running on the target machine:
  - Our examples are for x86 running Linux (same as vm for project)
  - Details vary slightly between different CPUs and OSs:
    - \textbf{Little endian (x86)} vs. big endian (Motorola)
    - Stack growth direction: \texttt{down} (x86 and most others)
    - Stack frame structure (OS and compiler dependent)
Linux process memory layout

- **%esp**
- **brk**
- **Loaded from exec**
- **unused**
- **shared libraries**
- **run time heap**
- **user stack**

- **0x08048000**
- **0x40000000**
- **0xC0000000**

- **LOW ADDR**
- **HIGH ADDR**
x86 __cdecl function-call convention

**Caller:**
```c
void foo(int a, int b, int c) {
    char buffer[5];
}
```
**asm:**
```
pushl $ebp
movl %esp, %ebp
subl $8, %esp
...  
```

<table>
<thead>
<tr>
<th>SP</th>
<th>0x00000003</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>0x00000002</td>
</tr>
<tr>
<td>SP</td>
<td>0x00000001</td>
</tr>
<tr>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>FP</td>
<td></td>
</tr>
<tr>
<td>Return address</td>
<td></td>
</tr>
<tr>
<td>Saved Frame Pointer</td>
<td></td>
</tr>
<tr>
<td>0x1F602BD1</td>
<td></td>
</tr>
<tr>
<td>0xAF6BA605</td>
<td></td>
</tr>
</tbody>
</table>

**Callee:**
```
call foo
```
x86 __cdecl function-call convention

• Push parameters onto the stack, from right to left
• call the function (pushes %eip+j to stack; return address)
• Save and update the FP (push %ebp + mov %esp,%ebp)
• Allocate local variables (sub $n,%esp)
• Perform the function's purpose
• Release local storage (add $n,%esp)
• Restore the old FP (leave = mov %esp,%ebp + pop %ebp)
• ret from function (pops return address and jumps to it)
• Clean up parameters (add $m,%esp)
Stack Frame

- Parameters
- Return address (ret)
- Saved Frame Pointer (sfp)
- Local variables

FP

SP

top of stack

HIGH ADDR

LOW ADDR
Smashing the stack

- Example of vulnerable function:

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

- When the function `foo` is invoked the stack looks like:

  ![Stack Diagram]

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

  ![Stack Diagram]
Return address clobbering

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

```
\text{exec("/bin/sh")} \text{ret} \text{str}
```

(top of stack)

(exact shellcode given by Aleph One)

• When foo returns, the user will be given a shell!
  – If web server calls foo() with given URL attacker can get shell by entering long URL in a browser!

• Attack executes data from the stack
  – x86 allows data on the stack to be executed as code
Exploiting buffer overflows

• Some complications:
  – Need to determine/guess position of ret
  – Shellcode should not contain the ‘\0’ character
  – Overflow should not crash program before foo() exists

• Remotely exploitable overflows by return address clobbering:
  – (2005) Overflow in MIME type field in MS Outlook

    Set test = CreateObject("Symantec.SymVAFFileQuery.1")
    test.GetPrivateProfileString "file", [long string]
Stack-based attacks: many variants

- Return address clobbering
- Overwriting function pointers (e.g. PHP 4.0.2, MediaPlayer BMP)
  - buf[128]
  - FuncPtr
- Overwriting exception-handler pointers (C++)
  - Need to cause an exception afterwards
- Overwriting longjmp buffers (e.g. Perl 5.003)
  - Mechanism for error handling in C
- Overwriting saved frame pointer (SFP)
  - Off-by-one error is enough: one byte buffer overflow!
  - First return (leave) sets SP to overwritten SFP
  - Second return (ret) jumps to fake top of stack
Buffer overflows

HEAP-BASED ATTACKS
Heap-based attacks

• Compiler generated function pointers (e.g. C++ code)

Suppose vtable is on the heap next to a string object:
Heap-based attacks

- Compiler generated function pointers (e.g. C++ code)

- After overflow of `buf` we have:

```
Object T
```

```
buf[256] vtable
```

```
shell code
```

```
ptr data
```

```
Object T
```
A reliable exploit?

```javascript
shellcode = unescape("\%u4343\%u4343\%...\";
overflow-string = unescape("\%u2332\%u4276\%...");

cause-overflow( overflow-string ); // overflow internal buf[ ]
</SCRIPT>
```

Problem: attacker does not know where browser places `shellcode` on the heap

```
buf[256]  vtable  shellcode
```
Heap Spraying  [SkyLined 2004]

Idea:
1. use Javascript to “spray” heap with shellcode (and NOP slides)
2. then point vtable ptr anywhere in spray area
var nop = unescape("%u9090%u9090")
while (nop.length < 0x100000) nop += nop

var shellcode = unescape("%u4343%u4343%..."_REGISTRY_OBJECT);

var x = new Array();
for (i=0; i<1000; i++) {
    x[i] = nop + shellcode;
}

• Pointing func-ptr almost anywhere in heap will cause shellcode to execute.
Vulnerable buffer placement

- Placing vulnerable `buf[256]` next to object O:
  - By sequence of Javascript allocations and frees make heap look as follows:
    - Allocate vulnerable buffer in Javascript and cause overflow
    - Successfully used against a Safari PCRE overflow [DHM’08]
## Many heap spray exploits

<table>
<thead>
<tr>
<th>Date</th>
<th>Browser</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/2004</td>
<td>IE</td>
<td>IFRAME Tag BO</td>
</tr>
<tr>
<td>04/2005</td>
<td>IE</td>
<td>DHTML Objects Corruption</td>
</tr>
<tr>
<td>01/2005</td>
<td>IE</td>
<td>.ANI Remote Stack BO</td>
</tr>
<tr>
<td>07/2005</td>
<td>IE</td>
<td>javaprxy.d11 COM Object</td>
</tr>
<tr>
<td>03/2006</td>
<td>IE</td>
<td>createTextRang RE</td>
</tr>
<tr>
<td>09/2006</td>
<td>IE</td>
<td>VML Remote BO</td>
</tr>
<tr>
<td>03/2007</td>
<td>IE</td>
<td>ADODB Double Free</td>
</tr>
<tr>
<td>09/2006</td>
<td>IE</td>
<td>WebViewFolderIcon setSlice</td>
</tr>
<tr>
<td>09/2005</td>
<td>FF</td>
<td>0xAD Remote Heap BO</td>
</tr>
<tr>
<td>12/2005</td>
<td>FF</td>
<td>compareTo() RE</td>
</tr>
<tr>
<td>07/2006</td>
<td>FF</td>
<td>Navigator Object RE</td>
</tr>
<tr>
<td>07/2008</td>
<td>Safari</td>
<td>Quicktime Content-Type BO</td>
</tr>
</tbody>
</table>

- **Improvements:** Heap Feng Shui [Sotirov ’07]
  - Reliable heap exploits **on IE** without spraying
  - Gives attacker full control of IE heap from Javascript

[RLZ’08]
Buffer overflows

Return-to-libc Attacks and Return-Oriented Programming
One more false assumption
Return-to-libc

• Control hijacking without code injection
  – Call library function (e.g. system) or dead code

• Remove security-sensitive functions from shared libraries?
  – this might break legitimate uses
Return-Oriented Programming

• When calling library/dead functions not helpful
  – e.g. if system is removed from libc.so

• Execute “opportunistic” code
  – Code in the middle of a function
  – Code obtained by jumping in the middle of instructions
    • x86 instructions are variable length

• Arbitrary(!) behavior without code injection
  – *if arbitrary jumping around within existing, executable code is permitted then an attacker can cause any desired, bad behavior, without code injection*
  – libc.so provides sufficiently large code base for this

• Reference: [Shacham et. al. ’07 & ‘09]
Ordinary programming (machine level)

- IP determines which instruction to fetch and execute
- IP incremented automatically after executing instr.
- Control flow (jumps) by changing IP
Return-oriented programming (machine level)

- SP determines which insns. to execute next
- SP incremented by the `ret` at the end of insns.
- Control flow (jumps) by changing SP (sub $n,%esp)
NOP

nop  nop  nop

instruction pointer

stack pointer

C library
ret
Load immediate constant

```
mov $0xdeadbeef, %ebx
```

```
0xdeadbeef
```

```
pop %ebx; ret
```

```
instruction pointer
```

```
stack pointer
```
Gadgets: multiple instruction sequences

- Example: load from memory into register
  - Load address of source word into %eax
  - Load memory at (%eax) into %ebx
Are there enough useful instruction sequences?

- In Linux libc, one in 178 bytes is a `ret (0xc3)`
  - One in 475 bytes is an opportunistic, or unintended, `ret`
Return-oriented compiler

- Generates shellcode given high-level exploit program
  
  ```
  var arg0 = "/bin/sh";
  var arg0Ptr = &arg0;
  var arg1Ptr = 0;
  trap(59, &arg0, &(arg0Ptr), NULL);
  ```

- Turing complete language
  - Sorting an array uses 152 gadgets, 381 instr. seq. (24 KB)

- No code injection!

- Not only on x86/CISC!
  - Also works on RISC (SPARC)
Return-oriented programming: workflow

- Connect back to attacker while socket not EOF
- Read line
- Fork, exec named progs

```
... again:
mov i(s), ch
cmp ch, 'l'
jeq pipe
... decr i
jnz again
... ?
load
cmp
ejq
```

**Stack:**
- `insns ... ret`
- `(data)`
- `insns ... ret`
- `(data)`
- `insns ... ret`
- `(data)`
- `insns ... ret`
INTEGER OVERFLOWS
Integer overflows

- Writing too large value into int causes it to “wrap around”
  - Assigning int to short
  - Arithmetic: int = int + int or int = int * int
- Example

```c
int table[800];

int insert_in_table(int val, int pos){
  if(pos > sizeof(table) / sizeof(int))
    return -1;
  table[pos] = val;
  // *(table + (pos * sizeof(int))) = val
  return 0;
}
```
Not always easy to exploit

- Example (OpenSSH 3.3)

```c
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp*sizeof(char*));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
```

- If nresp=1073741824 allocates a 0-byte buffer and overflows
Integer overflow stats

Source: NVD/CVE
FORMAT STRING VULNERABILITIES
Format string vulnerabilities

```c
int func(char *user) {
    printf(user);
}
```

- **Problem**: what if `user = "%s%s%s%s%s%s"` ??
  - Most likely program will crash: DoS.
  - If not, program will print memory contents. Privacy?
  - Full exploit if `user = "%n"

- **Correct form**:
  ```c
  int func(char *user) {
      printf("\"%s\", user);
  }
  ```
History

- First exploit discovered in June 2000.
- Examples:
  - wu-ftp 2.*: remote root
  - Linux rpc.statd: remote root
  - IRIX telnetd: remote root
  - BSD chpass: local root
  - ...
- Any function using a format string is vulnerable!
  - Printing: printf, fprintf, sprintf, …
  - Logging: syslog, err, warn
Exploiting

• Dumping arbitrary memory:
  – Walk up stack until desired pointer is found.
  – printf( "%08x.%08x.%08x.%08x|\%s|"")

• Writing to arbitrary memory:
  – printf( "hello %n", &temp) -- writes ‘6’ into temp.
  – printf( "%08x.%08x.%08x.%08x.%n")

• Read this for details:
  – Exploiting Format String Vulnerabilities, scut/team teso
Overflow using format string

```c
char errmsg[512], outbuf[512];
sprintf (errmsg, "Illegal command: %400s", user);
...  
sprintf( outbuf, errmsg );
```

- What if user = “%500d <nops> <shellcode>”
  - Bypass “%400s” limitation.
  - Will overflow outbuf, and get a shell
References

- Smashing The Stack For Fun And Profit, Aleph One
- Heap Feng Shui in JavaScript, Alexander Sotirov
- Return-Oriented Programming, Shacham et. al. 2009
- Basic Integer Overflows, blexim
- Exploiting Format String Vulnerabilities, scut/team teso
Project 1:

WRITING EXPLOITS
Project 1: writing exploits

- 7 vulnerable programs you need to exploit
  - should be increasingly difficult
  - buffer and integer overflows + format string vulnerabilities
- One practice target (target0)
  - Return address clobbering: should help you get started
  - will be exploited in the next tutorial
- Exploit skeletons provided + Aleph One’s shellcode
  - no need to write much code
  - will probably spend most time thinking, reading and debugging
- VMware virtual machine running Linux (Debian)
  - your exploits need to work in the vm
Project 1: writing exploits

- Teams of up to 2 people
  - if 2 people then should submit only one common set of exploits
- You get points only for successful exploits
  - need only 5 points for maximum grade, the rest are bonus
- The early bird catches the worm
  - additional bonus points for being the first to exploit a target
  - check status page first, if target still “available” send by email
- Hint #1: start early
- Hint #2: gdb is your friend
- Hint #3: use tutorials, office hours, bulletin board
Project 1: useful references

- Smashing The Stack For Fun And Profit, Aleph One
- Buffer overflows demystified, Murat
- The Frame Pointer Overwrite, klog
- Basic Integer Overflows, blexim
- Exploiting Format String Vulnerabilities, scut/team teso
- How to hijack the Global Offset Table with pointers for root shells, c0ntex
- Intel Architecture Guide for Software
HAVE FUN!