Control Hijacking Attacks

Buffer overflows and format string bugs

Buffer overflows

- Extremely common bug
- First major exploit: 1988 Internet Worm, 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 lead to total compromise of host.
- Developing buffer overflow attacks:
  - Locate buffer overflow within an application.
  - Design an exploit.

What is needed

- 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 CPUs and OSs:
    - Little endian vs. big endian (x86 vs. Motorola)
    - Stack Frame structure (Linux vs. Windows)
    - Stack growth direction.

Linux process memory layout

What are buffer overflows?

- Suppose a web server contains a function:
  ```c
  void func(char *str) {
      char buf[128];
      strcpy(buf, str);
      do-something(buf);
  }
  ```
- When the function is invoked the stack looks like:
  ```c
  high
  low
  ```
- What if *str is 136 bytes long? After `strcpy`:
  ```c
  low
  high
  ```
Basic stack exploit

Main problem: no range checking in strcpy().
Suppose *str is such that after strcpy stack looks like:

```
| 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.

Exploiting buffer overflows

Suppose web server calls func() with given URL:
ATTacker sends a 200 byte URL. Gets shell on web server.

Some complications:
- Program P should not contain the "\0" character.
- Overflow should not crash program before func() exists.

Sample remote buffer overflows of this type:
- Old overflow in MIME type field in MS Outlook.
- Old overflow in Symantec Virus Detection
  ```
  Set test = CreateObject("Symantec.SymVAFileQuery.1")
  test.GetPrivateProfileString "file", [long string] 
  ```

Control hijacking opportunities

Stack smashing attack:
- Override return address in stack activation record by overflowing a local buffer variable.

Function pointers: (e.g. PHP 4.0.2, MS MediaPlayer Bitmaps)
- Overflowing buf will override function pointer.

Longjmp buffers: ```longjmp(pos)``` (e.g. Perl 5.003)
- Overflowing buf next to pos overrides value of pos.

Other types of overflow attacks

Integer overflows: (e.g. MS DirectX MIDI Lib) Phrack80
```
void func(int a, char c) {
    char buf[128];
    init(buf);
    buf[3*a+1] = c;
}
```

Problem: 3*a+1 can point to "ret-addr" on stack.

Double free: double free space on heap.
- Can cause memory mgr to write data to specific locations.
- Examples: CVS server

Finding buffer overflows

To find overflow:
- Run web server on local machine.
- Issue requests with long tags.
- All long tags end with "$\ldots\$".
- If web server crashes, search core dump for "$\ldots\$" to find overflow location.

Some automated tools exist. (e.g. eEye Retina)
- Then use disassemblers and debuggers (e.g IDA-Pro) to construct exploit.
Preventing overflow attacks

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)

Marking stack as non-execute

Basic stack exploit can be prevented by marking stack segment as non-executable.
• NX-bit on AMD Athlon 64,  XD-bit on Intel P4 "Prescott".
  - NX bit in every Page Table Entry (PTE)
• Support in SP2.  Code patches exist for Linux, Solaris.

Limitations:
• Does not defend against `return-to-libc` exploit.
  - Overflow sets ret-addr to address of libc function.
• 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).

Static source code analysis

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.

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.

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.
StackGuard variants - ProPolice

ProPolice (IBM) - gcc 3.4.1. (-fstack-protector)
- Rearrange stack layout to prevent ptr overflow.

String Growth
- No arrays or pointers
- Args
- Ret addr
- SFP
- CANARY
- Arrays
- Local variables

Stack Growth
- Ptrs, but no arrays

Run time checking: Libsafe

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.

Windows XP SP2 /GS

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.

Format string problem

```c
int func(char *user) {
    fprintf(stdout, user);
}
```

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

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

- Danger discovered in June 2000.
- Examples:
  - wu-ftpd 2.*: remote root
  - Linux rpc.statd: remote root
  - IRIX telnetd: remote root
  - BSD chpss: local root

Vulnerable functions

Any function using a format string.

Printing:
- printf, fprintf, sprintf, ...
- vprintf, vfprintf, vsprintf, ...

Logging:
- syslog, err, warn

Exploit

- 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")

Overflow using format string

- 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.