Buffer Overflow Attacks and Format String bugs

Buffer overflows

- Extremely common bug.
  - First major exploit: 1988 Internet Worm, fingered.
- 10 years later: over 50% of all CERT advisories:
  - 1997: 16 out of 28 CERT advisories.
- Often leads to total compromise of host.
  - Fortunately: exploit requires expertise and patience.
- Two steps:
  - Locate buffer overflow within an application.
  - Design an exploit.

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

Basic stack exploit

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

Some unsafe C lib functions

```c
strcpy (char *dest, const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf (const char *format, ...)
printf (const char *format, ...)
```
Causing program to exec attack code

- Stack smashing attack:
  - Override return address in stack activation record by overflowing a local buffer variable.
- Function pointers: (used in attack on Linux superprobe)
  - Overwrite return address in stack activation record by overflowing a local buffer variable.
- Longjmp buffers: longjmp(pos) (used in attack on Perl 5.003)
  - Overwrite return address in stack activation record by overflowing a local buffer variable.

Finding buffer overflows

- Hackers find buffer overflows as follows:
  - Run web server on local machine.
  - Issue requests with long tags.
  - If web server crashes, search core dump for "$\ldots\ldots\ldots\$$" to find overflow location.
- Some automated tools exist. (eEye Retina, ISIC).

Preventing buf overflow attacks

- Main problem:
  - strcpy(), strcat(), sprintf() have no range checking.
  - "Safe" versions strncpy(), strncat() are misleading
    - strncpy() may leave buffer undertermined.
    - strncat() encourages off-by-1 bugs.
- Defenses:
  - Type safe languages (Java, ML). Legacy code?
  - Mark stack as non-executable. Random stack location.
  - Static source code analysis.
  - Run time checking: StackGuard, Libsafe, SafeC, (Purify).
  - Black box testing (e.g. eEye Retina, ISIC).

Marking stack as non-execute

- Basic stack exploit can be prevented by marking stack segment as non-executeable or randomizing stack location.
- Code patches exist for Linux and Solaris.
- Problems:
  - Does not block more general overflow exploits:
    - Overflow on heap: overflow buffer next to function pointer.
  - Some apps need executable stack (e.g. Lisp interpreters).
- Patch not shipped by default for Linux and Solaris.

Static source code analysis

- Statically check source to detect buffer overflows.
  - Several consulting companies.
- Can we automate the review process?
- Several tools exist:
  - @stake.com (Ophit.com): SLINT (designed for UNIX)
  - retcorpi: its4. Scan function calls.
- Find lots of bugs, but not all.

Run time checking: StackGuard

- Many many run-time checking techniques...
- 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: 4% for Apache.

- Newer version: PointGuard.
  - Protects function pointers and set jmp 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.

---

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.

Run time checking: many others …

- Address obfuscation, (Stan Brock '03)
  - Encrypt return address on stack by XORing with random string. Decrypt just before returning from function.
  - Attacker needs decryption key to set return address to desired value.

- Randomize location of functions in libc.
  - Attacker cannot jump directly to exec function.

---

Format string problem

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

Problem: what if user = "%s%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) {
    printf(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 chpass`: 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

```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`.

Timing attacks

- Timing attacks extract secret information based on the time a device takes to respond.

- Applicable to:
  - Smartcards.
  - Cell phones.
  - PCI cards.
Consider the following pwd checking code:

```c
int password-check(char *inp, char *pwd)
    if (strlen(inp) != strlen(pwd)) return 0;
    for( i=0; i < strlen(pwd); ++i)
        if (*inp[i] != *pwd[i])
            return 0;
    return 1;
```

A simple timing attack will expose the password one character at a time.

Alternative code:

```c
int password-check(char *inp, char *pwd) {
    oklen = (strlen(inp) == strlen(pwd)) ;
    for( ok=1, i=0; i < strlen(pwd); ++i)
        if (*inp[i] != *pwd[i])
            return 0;
    return ok & oklen;
}
```

Timing attack is ineffective ... (?)