Gliederung Programmiersprachen

1. Von der Maschinenmaschinen- sprache zu C
2. Die Struktur von C-Programmen
3. Variable und Datentypen in C
4. Bildschirm Ein-/Ausgabe
5. Kontrollstrukturen
6. Funktionen
7. Programmierstil, Programmierhilfsmitteln
8. Felder u. Zeichenketten
9. Ausdrücke
10. Arbeiten mit Daten
11. Strukturen, Autzähllungstypen
12. Zeiger
13. Enum, Strukturen und Unions
14. Speicherklassen
15. Vertiefung einiger Themen: Sicherheit

Buffer Overflow- und Formatstring-Angriffe

Buffer Overflow Attacks
and Format String bugs

Quelle:
http://crypto.stanford.edu/cs155/syllabus.html

What are buffer overflows?

- Suppose a web server contains a function:

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

- When the function is invoked the stack looks like:

```
top
| pf  | \n| \n| \n| \n| \n
```

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

```

top
| pf  | \n| \n| \n| \n| \n```

Basic stack exploit

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

```
| pf  | \n| \n| \n| \n| \n```

Program P: `exec( "/bin/sh" )`

\( ( \text{exec shell code by A Leigh Orla) } \)

- 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.

Buffer overflows

- Extremely common bug.
  - First major exploit: 1988 Internet Worm. `fingerd`.
  - 10 years later: over 50% of all CERT advisories:
    - 1997: 16 out of 28 CERT advisories.
    - 1998: 9 out of 13 `--`
    - 1999: 6 out of 12 `--`
- Often leads to total compromise of host.
  - Fortunately: exploit requires expertise and patience.
  - Two steps:
    - Locate buffer overflow within an application.
    - Design an exploit.
Some unsafe C lib functions

- strcpy (char *dest, const char *src)
- strcat (char *dest, const char *src)
- gets (char *s)
- scanf (const char *format, …)
- printf (const char *format, …)

Exploiting buffer overflows

- Suppose web server calls `fcntl()` with given URL.
- Attacker can create a 200 byte URL to obtain shell on web server.

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

- Sample buffer overflows of this type:
  - Overflow in MIME type field in MS Outlook.
  - Overflow in ISAPI in IIS.

Preventing buf overflow attacks

- Main problem:
  - `strcpy()`, `strcat()`, `sprintf()` have no range checking.
  - “Safe” versions `strncpy()`, `strncat()` are misleading
  - `strncpy()` may leave buffer unterminated.
  - `strncpy()`, `strcat()` 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).
  - 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-executable 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 func pointer.
  - Some apps need executable stack (e.g. LISP interpreters).
  - Patch not shipped by default for Linux and Solaris.

Finding buffer overflows

- Hackers find buffer overflows as follows:
  - Run web server on local machine.
  - Issue requests with long tags.
  - All long tags end with “$$$$$”.
  - If web server crashes, search core dump for “$$$$$” to find overflow location.

- Some automated tools exist. (eEye Retina, ISIC).

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 `supercache`

- Buffer [0] Stack

- Buffer next to `func` will override function pointer.

- Longjmp buffers: `longjmp(pos)` (used in attack on Perl 5.03)
  - Overflowing buffer next to `pos` overrides value of `pos`.
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 (O2R.com): SLINT (designed for UNIX)
- 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: 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.

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 …

- Address obfuscation. (Stony Brook ’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 bugs

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 = "%if"`

Correct form:

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

History

- Danger discovered in June 2000.
- Examples:
  - `wu-ftpd 2.0`: 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`.

```c
// For example, use %n instead of %0x.%0x.%0x.%0x.%n`
Timing attacks

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

- Applicable to:
  - Smartcards.
  - Cell phones.
  - PCI cards.

Timing attacks: example

Consider the following pwd checking code:

```c
int password_check(char *rn, char *pwd)
{
    if (strcmp(rn, pwd) != 0) return 0;
    for (i=0; i<strlen(pwd); i++)
        if (*rn++ != *pwd++)
            return 0;
    return 1;
}
```

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

Timing attacks: example

Correct code:

```c
int password_check(char *rn, char *pwd)
{
    oklen = (strlen(rn) == strlen(pwd));
    for (cl=1, i=0; i<strlen(pwd); i++)
        if (rn[i] == pwd[i])
            cl = cl & (1 << i);
    return cl & oklen;
}
```

Timing attack is ineffective ...