ELEC5616 COMPUTER & NETWORK SECURITY

Lecture 22:
Software Security

DESIGN OF A SECURE SYSTEM

Top Down Approach:

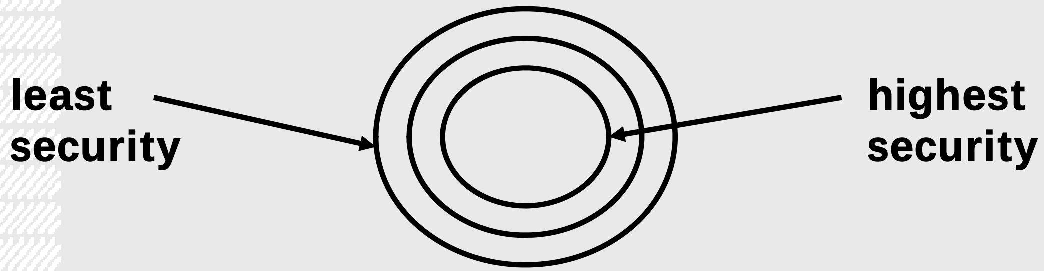
1. **Threat Model -** What are the likely risks?
2. **Security Policy -** What is the security meant to achieve?
3. **Security Mechanisms -** How are the mechanisms implemented?

Design of a secure system follows a ring design

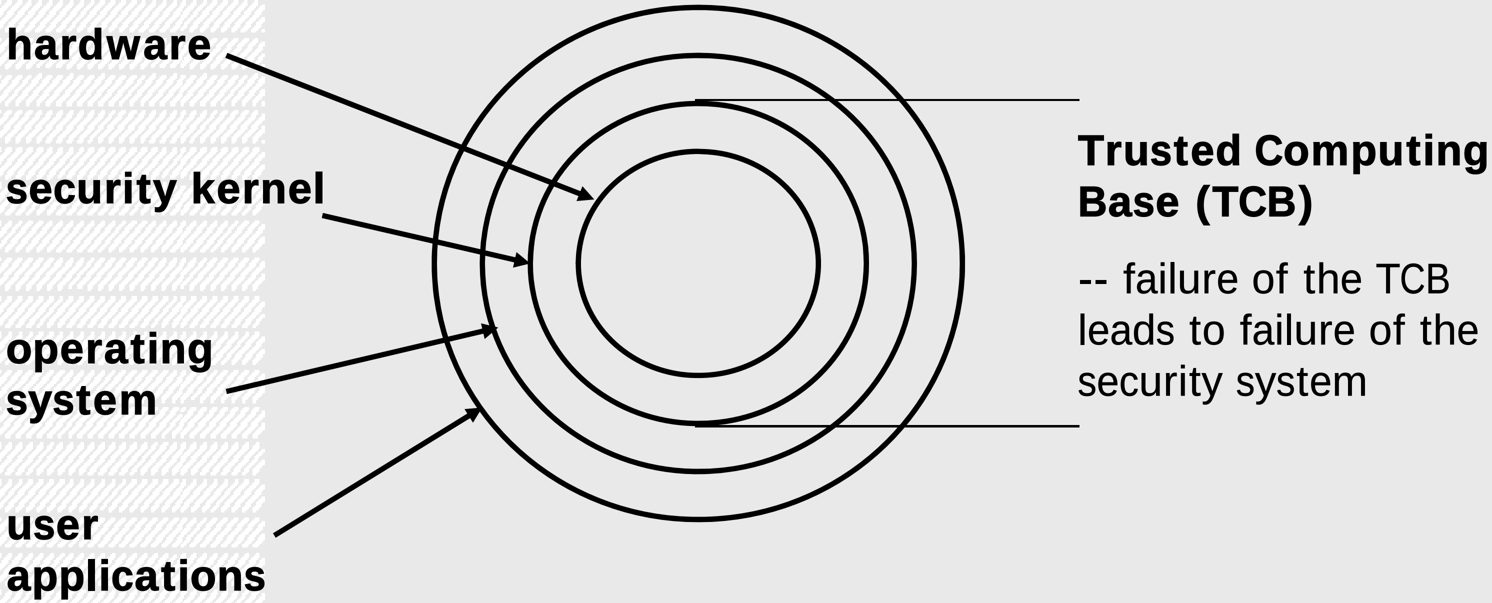
Every object has an associated security attribute

Every subject has a security clearance

The aim is to restrict the interaction between rings



EXAMPLE: TRUSTED OPERATING SYSTEM



BELL - LAPADULA MODEL

Developed in the context of military and intelligence data

Also known as “multilevel security” or “MLS systems”

Simple Security Property:

- No process may read data at a higher level (no read up - NRU)
- Agent Alice with “Secret” clearance can’t read Top-Secret documents

The *-property:

- No process may write data to a lower level (no write down- NWD)
- Agent Alice with “Secret” clearance can’t write Public documents

COVERT CHANNELS

Bell-LaPadula: prevent subjects with different access rights from communicating.

- Problem: covert channels.

Covert channels:

- Communication channels undetected by the security policy enforcer.

Example: File locking:

- High clearance subject frequently locks and unlocks a file.
- Low clearance subject checks lock status.
- Using synchronized timer: 1000bit/sec transfer rate.

EVALUATION: THE ORANGE BOOK

Trusted Computer Systems Evaluation Criteria

US Department of Defense, 1979

Ratings

D: Minimal protection. Anyone can get this rating.

C1: Discretionary security. Users can disable the security

C2: Controlled access. Per user protection. Discretionary.

B1: Labeled protection. Every object is labeled. Bell-LaPadula

B2: Structured protection. More OS module verification.

B3: Security domains. Modular OS design. Clear security policy.

A1: Verified design. Formally verified system design.

Example: Windows NT is considered C2 compliant.

(http://csrc.nist.gov/publications/history/dod85.pdf)

COMMON PROGRAMMING MISTAKES

Poor design choices / assumptions
Failing to check or escape user supplied input
Buffer overflows
Incorrect levels and separation of privileges
Poor default settings
Predictable sources of randomness
Race conditions
Poor failure modes
Insufficient testing
Failure to protect secrets properly
Poor design review
Poor documentation and commenting
Poor maintenance of legacy code
Bad programming languages

BUFFER OVERFLOWS

One of the most common bugs today

- Packet Storm has ~ 100 new ones every month

Mainly due to poor programming practice, poor languages from a security perspective (e.g. C) and lack of proper code review.

Easy to exploit :

- Find problem in the source code
- Craft an exploit

Today there exists even automatic exploit generators that will drop in the right shellcode for a particular processor and operating system.

BUFFER OVERFLOWS EXPLAINED

Suppose a server contains the function `foo`:

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

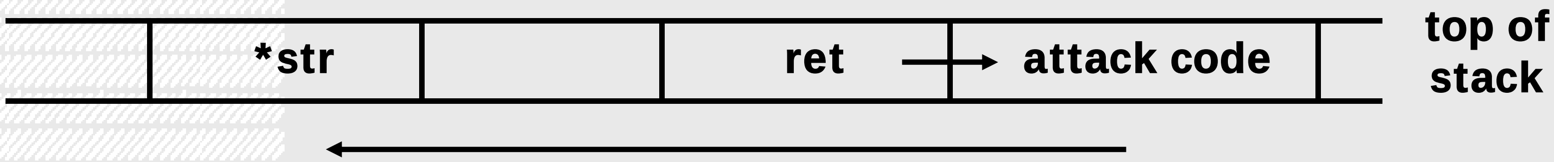
When the function is invoked the stack looks like:

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:



Attack code: execv("/bin/sh", 0)

When the function exits, it returns using the new return address and starts executing the attack code - giving a shell.

Note: the attack code runs in the stack.

EXPLOITING BUFFER OVERFLOWS

Suppose the above code is from a web server and `foo()` is called with a URL sent by the browser.

An attacker can create a 200-byte URL to obtain shell on web server.

Some complications:

- The attack code should not contain the '\0' character.
- Overflow should not crash program before foo() exists.

Recent buffer overflows of this type (www.us-cert.gov):

- **TA07-089A: Microsoft Windows Animated Cursor Buffer Overflow (March 2007)**
  A buffer overflow vulnerability in the way Microsoft Windows handles animated cursor files is actively being exploited.
- **TA05-362A: Microsoft Windows Metafile Handling Buffer Overflow (December 2005)** Microsoft Windows is vulnerable to remote code execution via an error in handling files using the Windows Metafile image format.

MORE GENERAL EXPLOITS

Basic stack exploit can be prevented by marking the stack segment as non-executable:

- Various solutions exist but there are many ways to trick
- Does not block more general overflow exploits.

General buffer overflow exploits are based on two steps:

- Arrange for attack code to be present in program space.
- Cause program to execute attack code.

INSERTING THE ATTACK CODE

Injecting attack code:

Place code in stack variable (local variables).

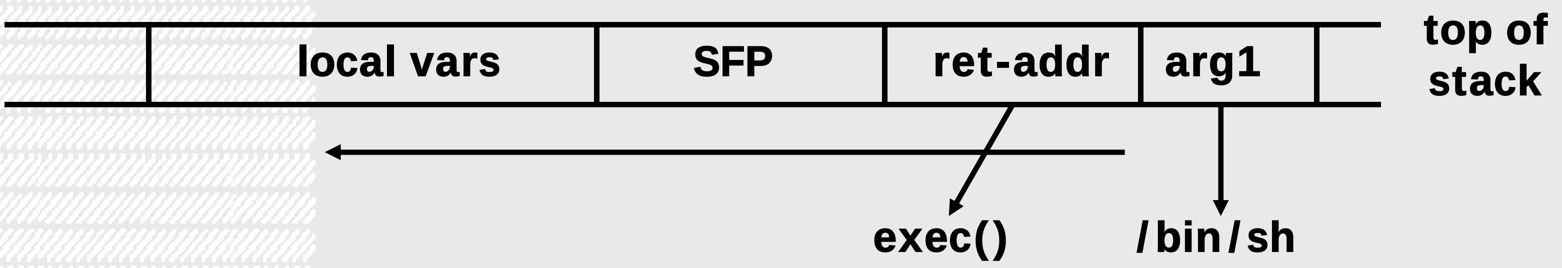
Place code in a heap variable (malloc'd variables).

Place code in static data segment (static variables).

Using existing code: return-into-libc (exec)

Cause function pointer or return address to point to the libc "exec" function.

At same time override first argument to be "/bin/sh"



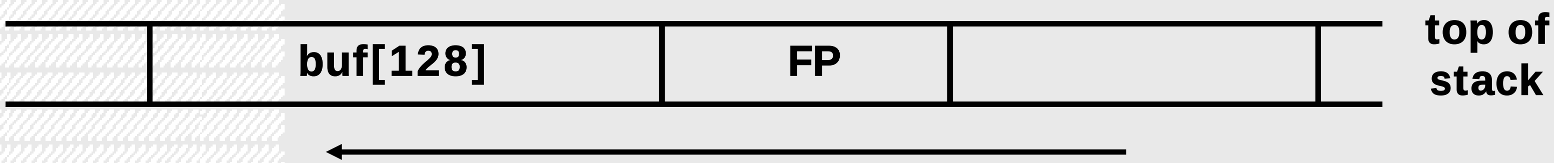
CAUSING EXECUTION OF ATTACK CODE

Stack smashing attack:

Override return address in stack activation record by overflowing a local buffer variable.

Function pointers:

Overflowing buf will override function pointer.



Longjmp buffers: **longjmp(pos)**

Overflowing buf next to pos overrides value of pos.

FINDING BUFFER OVERFLOWS

Hackers find buffer overflows as follows:

- Run software on local machine.
- Supply long strings into user input fields. Long strings are of the form “XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX”.
- If the software crashes, search the core dump for “XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX” to find the overflow location.

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

PREVENTING BUFFER OVERFLOWS

Main problem:

- strcpy(), strcat(), sprintf() have no range checking.
- "Safe" versions strncpy(), strncat() are often misleading
  - strncpy() may leave buffer unterminated.
  - strncpy(), strncat() encourage off by 1 bugs.

strncpy(dest, src, strlen(src)+1)

Defenses:

1. Static source code analysis.
2. Run time checking.
3. Black box testing (e.g. eEye Retina, ISIC).

STATIC SOURCE CODE ANALYSIS

Statically check source to detect buffer overflows.

- Internal code reviews
- Consulting companies ($$$)

Automated tools

- Still requires expertise to run and analyse results
- @stake.com (l0pht.com): SLINT (designed for UNIX)
- rstcorp: its4. Scans function calls.
- Berkeley: Wagner, et al. Tests constraint violations.

Also: things like RIPS for PHP

RUN TIME CHECKING

Runtime range checking

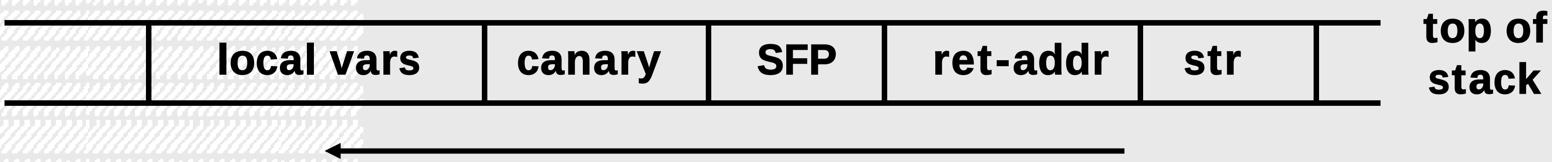
Significant performance degradation.

Hard for languages like C and C++.

StackGuard (OGI)

Run time tests for stack integrity.

Embed “canaries” in stack frames and verify their integrity prior to returning from the function.



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, new line, linefeed, EOF.

String functions will not copy beyond terminator.

Hence, attacker cannot use string functions to corrupt stack.

Other functions still have problems.

PROPOLICE (SSP)

ProPolice implemented as a GCC patch, now in GCC 4.1.

Minimal performance effects.

Protects more than just return address

Note: Canaries don’t offer foolproof protection.

- Some stack smashing attacks can leave canaries untouched.

Related: Address Space Layout Randomisation (ASLR)

- Randomise placement of certain code in process’ address space

RACE CONDITIONS

A <u>race condition</u> attack exploits the fact that multiple instruction transactions are not atomic

<u>Example:</u>

Race condition in the old UNIX “`mkdir`” command

`mkdir` executed with two Stages:

- Storage allocated
- Ownership transferred to user

Attack:

- User initiates `mkdir`
- User quickly replaces the new directory with /etc/passwd
- `mkdir` process transfers ownership to user

Many such problems have existed with temporary files in /tmp

RACE CONDITIONS

Example:

Print server race condition with print quotas

- User prints file
- Print server determines cost of file
- Print server checks account balance and deducts amount
- Print server spools file for printing

Attack:

```
lpr smallfile
sleep (1)
cat bigfile > smallfile
```

TIMING ATTACKS

Timing attacks extract secret information based on the time a device takes to respond ("side channel attack")

Applicable to:

- Smartcards
- Cell phones
- PCI cards
- Network software

Examples:

- RSA exponentiation
- Password checking and lengths
- Inter-keystroke timing (e.g. attack on ssh)

TIMING ATTACKS

Consider the following password checking code:

```
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 length and the password one character at a time.

TIMING ATTACKS

Correct code:

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

Timing attack is ineffective.

REFERENCES

Papers

Smashing the Stack for Fun and Profit (Aleph One)

http://www.insecure.org/stf/smashstack.txt

Timing Analysis of Keystrokes and Timing Attacks on SSH (Wagner et al)

http://www.usenix.org/events/sec01/song.html