<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:30 – 11:20</td>
<td>Lecture I</td>
<td>DTC</td>
</tr>
<tr>
<td>11:20 – 12:40</td>
<td>Lunch</td>
<td>DTC</td>
</tr>
<tr>
<td>12:40 – 2:30</td>
<td>Lecture II</td>
<td>DTC</td>
</tr>
<tr>
<td>3:10 – 5:00</td>
<td>Lab</td>
<td>Lind 24</td>
</tr>
</tbody>
</table>
THINKING LIKE AN ADVERSARY
TERMINOLOGY

• A **security goal** is a property or invariant the system attempts to maintain.

• An **attack** is an attempt to violate the security goals of a system.

• A **vulnerability** is a “hole” that allows an attack to succeed.

• A **weakness** is “almost” a vulnerability.
SECURITY ASSESSMENT

Confidentiality?

Integrity?

Availability?

Accountability?

Dependability?

“Security by Obscurity:” a system that is only secure if the adversary doesn’t know the details is not secure!
RULES OF THUMB

Be **conservative**: evaluate security under the best conditions for the adversary.

A system is as secure as the **weakest link**.

It is best to plan for **unknown** attackers.
SOFTWARE SECURITY

Software is “secure” if it correctly performs its intended task in the presence of an adversary.
CONTROL HIJACKING
(or, Why to Avoid C Like the Plague)

A control hijacking attack injects new code into a running process.

There are many ways to hijack a C program.

Most common is the buffer overflow: writing outside the bounds of a chunk of memory.
C functions store local variables on a “stack” including arguments and return address.

Operating systems maintain virtual memory, not user programs. The stack, code, constants, etc. always have the same address.

Running programs are stored in memory. The same code can have many stack frames.

The C standard libraries have many ways to run other programs – exec*, system, popen...
THE STACK FRAME

Parameters
Return address
Stack Frame Pointer
Local variables

SP

Stack Growth
BUFFER OVERFLOW IDEA

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

Suppose we call `func("aa...aaBCDE")`

**On enter:**

<table>
<thead>
<tr>
<th>(buf)</th>
<th>sfp</th>
<th>ret-addr</th>
<th>str</th>
</tr>
</thead>
</table>

**After `strcpy`:**

| aa..aa | a..a | BCDE | X |

**On return:**

`sp = aaaa; jmp BCDE`
The “classic” stack exploit (e.g. from @1) sets *str so that after strcpy, we have:

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

return from `func(str)` jumps to our code for P, and gives the (possibly remote) user a shell.
HOW TO CONSTRUCT *str?

Given the source code, we can use a debugger. Break at func, print &buf, to get the address.

With no source, try guessing stack depth. At most 4B possible starting points.

Computers are fast.

Code for P: Use a compiler, or google “shellcode”
UNSAFE LIBRARY FUNCTIONS

strcpy (char *dest, const char *src)
strcat (char *dest, const char *src)
ggets (char *s)
scanf (const char *format, ...)
sprintf (conts char *format, ...)
...
HIJACKING METHODS

Control Flow can be hijacked in several ways:

- **Stack Smashing** (changing the return address)

  [http://insecure.org/stf/smashstack.html](http://insecure.org/stf/smashstack.html)

- Modifying **function pointers** (on the heap, in the stack, or in the “Global Offset Table”)

  [http://www.milw0rm.com/papers/3](http://www.milw0rm.com/papers/3)

- Modifying **setjmp/longjmp buffers.**

  [http://www.w00w00.org/files/articles/heaptut.txt](http://www.w00w00.org/files/articles/heaptut.txt)
FINDING OVERFLOWS

1. Obtain local copy of target software. (e.g. web server)

2. Run on long, distinctive inputs, until program dumps core.

3. Search core dump for inputs to find overflow location.

B) Use automated tool. (google)

C) BugTraq.
Buffer overflows happen because C lacks array bounds. So **Don’t Use C!**

Stuck with C?

- Non-executable stack
- Randomization
- Source Code Analysis
- Run-time tools
NONEXECUTABLE STACK OVERFLOWS

The “famous” stack-smashing attack puts code on the stack, so fails if the stack is not executable.

If goal is, e.g., to get a shell, no particular reason we need to call our own code. Instead:

```
exec ... /bin/sh sfp ret arg
```

Return from `func(str)` calls `exec("/bin/sh")`
RANDOMIZATION

Address space randomization: change locations of stack, heap, data segment, shared libs...

return-to-libc has to find it first...

Instruction set randomization: change opcodes on machine. Supported by some processors!

injected code usually won’t run...

Both make exploits harder, but...

www.usenix.org/events/sec05.tech/full_papers/sovarel/sovarel.pdf
SOURCE CODE ANALYSIS

It is undecidable to determine, from code, whether a program has a buffer overflow.

Some “rules of thumb” let us look for places where overflows are possible:

Look for common bugs – off by one, “bad” library calls, memory leaks, etc...

http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx

Constraint violations: propagate bounds and look for possible out-of-bounds accesses.

http://www.cs.berkeley.edu/~daw/papers/overruns-ndss00.ps

“Trust inconsistencies” – assign trust values to uses, buffers. Look for mismatches.

www.coverity.com/
RUN TIME CHECKING I

Check that library calls are safe at runtime.

e.g. Libsafe:
Intercepts calls to e.g. `strcpy(dest, src)`
  – Check for space in current stack frame:
    \[
    \text{\mid frame-pointer} - \text{dest} \mid > \text{strlen(src)}
    \]
  – If so, does proceed.
  Otherwise, terminate application.
Coal Miner approach: watch for “canary” to change and when it does, get out fast!

StackGuard, ProPolice, XP SP2 /GS option...
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

- GCC patch, use random canaries
  - 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 foolproof protection.
  - Some stack smashing attacks can leave canaries untouched.
FORMAT STRING BUGS

```c
int func(char *user_input) {
    printf( user_input );
}
```

Problem: what if `user = "%s%s%s%s%s%s%s%s"`?

Most likely program will crash...

If not, program will print memory contents.

Correct form:

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

char outbuf[512], errmsg[512];
sprintf(errmsg, "Illegal cmd: %.400s", input);
sprintf(outbuf, errmsg);

How could we use format strings to overflow outbuf?

input = "%512x|Any 394 byte overflow value"
Important things about format strings:

\[
\text{printf(“\%4$c”, ‘a’, ‘b’, ‘c’, ‘d’)} \text{ prints ‘d’.}
\]

\[
\text{printf(“\%42x\%n”, 0, \&var)} \text{ sets var=42.}
\]

Suppose we want to change the byte at address ADDR to VALU, and know that the buffer with the format string is POS-4 words up the stack:

\[
\text{buf = “\%VALUx\%POS$hhnxxxADDR”}
\]
PREVENTING FORMAT-STRING BUGS

Correct usage, input validation, snprintf

FormatGuard: count arguments to printf, parameters in format string.
www.usenix.org/events/sec01/cowanbarringer.html

Tainting: reject format strings from “untrusted” sources, strings with %n, etc...

Whitelisting: allow writes only to program-addressable locations.
www.cs.washington.edu/homes/miker/format_string.pdf
COMMON SECURITY BUGS

Like buffer overflows, the most common reason for security bugs is invalid assumptions.

An adversary will look for these assumptions and find ways to invalidate them.
WEAK INPUT CHECKING

Adversaries can often control program inputs:
– Direct input: command line, keyboard, …
– Function calls
– Config files
– Network packets
– Web forms…

Bug: it is common to assume input is “benign”
EXAMPLE: system()

Web forms are often processed by scripts that need to run other programs on the server. Usually scripts use C’s system() or popen().

Bug: these calls invoke a shell. Command separators like “|” and “;” allow the user to run other commands on the server.

Form.cgi:
# ... start html ...
system("grep $test file");
# ... do other stuff ...

Attacker inputs:
“; cat /etc/passwd”

Web Server runs:
“grep ; cat /etc/passwd file”
IIS has the security goal that only commands in the subdirectory /scripts should be executed.

So it checks that the URL matches /scripts/*.*


IIS tries to fix this by filtering out URLs with “../” in them, before unicode expansion.

http://a.b.c.d/scripts/%c0%af..%c0%afwinnt/system32/cmd.exe?X
void caller() {
    unsigned int a = read_int_from_network();
    char *z = read_string_from_network();
    if (a > 0) callee(a,z);
}

void callee(int a, char* z) {
    char buffer[10]; // … do some other stuff…
    for(int i = 0; i + a < 10 && z[i]; i++)
        buffer[a+i] = z[i];
    return;
}

Bug: what if \(a = 2^{32}-10\)?
A **race condition** occurs when there is a nonzero time interval between checking some property and when it needs to hold true.

Race conditions are often called Time of Check / Time of Use (TOCTOU) vulnerabilities.
EXAMPLE

- Ghostscript creates a lot of temporary files:
  
  ```
  name = mktemp("/tmp/gs_XXXXXXXX");
  fp = fopen(name,"w");
  ```

- Attacker creates symlink
  
  `/tmp/gs_12345A -> /etc/passwd`
  
  between call to mktemp and fopen, when root is running gs (just has to happen once!)

- Ghostscript (as root) overwrites `/etc/passwd`. 
DIEBOLD CASE STUDY

Feldman et al. demonstrated exploitable vulnerabilities in the AccuVote-TS DRE system.

On reboot, the terminal checks CF card for the bootloader.

If `fboot.nb0` exists, boot from (unauthenticated) CF card.

The CF card slot is protected by a hotel mini-bar key.

CF Cards are used to upload ballot definitions, and download results.
DEFENSIVE PROGRAMMING

- Security Design Principles
  - Saltzer & Schroeder (1975)
- Programming “best practices”
SALTZER & SCHROEDER PRINCIPLES

- Economy of Mechanism
- Fail-safe defaults
- Complete Mediation
- Open Design
- Separation of Privilege/Least Privilege
- Least Common Mechanism
- Defense in Depth
- Psychological Acceptability
- Work Factor
- Compromise Recording
Which is harder:
Breaking encryption approved by NSA
Finding buffer overflow in proprietary software
Guessing passwords
V&M: PROMOTE PRIVACY

- A system should only give out information that is necessary.

Examples:
- Web servers & credit cards: even if they are stored, do not reveal to customers.
- Remote logins: no reason to reveal OS, etc. before authentication.
- “Finger” bugs: no reason to reveal the users of a system to others.
V&M: TRUST NO ONE

• Security is hard. Security-critical systems should not be trusted without extensive reviews.
  – Don’t use nonstandard crypto
  – Don’t reinvent the wheel
  – Externally review internal code – don’t trust yourself!

• Transitive trust: if component B can cause security of A to fail, and component C can cause B to fail, A must trust C.
DEFENSIVE PROGRAMMING

Best practices:
• Modular Design
• Check error conditions
• Validate inputs: whitelist vs blacklist
• Avoid infinite loops, memory leaks
• Check for integer overflows
• Language/library choices
• Development processes
EXAMPLES

char charAt(char *str, int index) {
    return str[index];
}

char *double(char *str) {
    size_t len = strlen(str);
    char *p = malloc(2*len+1);
    strcpy(p, str);
    strcpy(p+len, str);
    return p;
}
```c
typedef struct link {
    char *data;
    struct link *next;
} list;

in_place_merge(list *input1, list *input2) {
    list *output = NULL, *tail, *next;
    do {
        if(strcmp(input1->data, input2->data) <= 0) {
            next = input1;
            input1 = input1->next;
        } else {
            next = input2;
            input2 = input2->next;
        }
        if (output == NULL) {
            output = tail = next;
        } else {
            tail->next = next;
            tail = tail->next;
        }
    } while (input1 != NULL && input2 != NULL);
    if (input1 != NULL) tail->next = input1;
    else tail->next = input2;
    return output;
}
```
MODULAR DESIGN

• System should be broken down into modules:
  – Clear functionality: less chance of mental errors by caller
  – Clean interfaces: decrease possible interactions
• Least Privilege at the module level
  – E.g. inetd wrapper
  – E.g. web server
• Isolate modules: use language tools, system processes..., etc.
In languages without exceptions:
- Check “error conditions” on return values
- E.g. malloc(), open(), etc...

Catch exceptions or declare them

Think about where to handle errors
- Fix locally
- Propagate to caller
- Fail-stop
INPUT VALIDATION

• Before using an input value check that it is safe:
  – NULL, Out of range, invalid format, too long, too short, etc...

• Err on the side of caution:
  – I know this will be safe vs
  – I can’t think of a way to break this...

• E.g. whitelist safe inputs, rather than blacklist dangerous ones.
LOOPS & MEMORY LEAKS

- Check preconditions for a loop
- Sanity check return values of functions
- Prefer (safe) exit with error condition to “muddle through”
- Avoid algorithm denial-of-service
  - Eg. Hash table with O(n) worst-case
- Safe exit:
  - Free allocated heap objects
  - Maintain consistent state
  - Use error conditions
INTEGER OVERFLOWS

• Mismatch with programmer “mental model”
• Check for them!
  – Check inputs in proper range
  – Check \((a+b)\) for overflow
  – Watch out for implicit casting, sign extension…
• Test corner cases: \(-1, 0, 1, 2^{31}-1, -2^{31} \ldots\)
Language and library make a difference
- C lets you overwrite object boundaries
- C libraries encourage this
If you can, use a type-safe language: Java, C#, Ada, ML, Python, Erlang...
If you can’t, consider safer string-handling, I/O libraries: CCured, Purify, PREfix, PREfast, ...
DEVELOPMENT/TESTING PROCESS

- Design for security
- Use pre-, post- conditions, invariants
- Do code reviews
- Test cases:
  - Long inputs
  - Format specifiers, newlines, NULs…
  - Unprintable characters
  - Extreme values
  - Malformed inputs: aliased or overlapping pointers, cyclic structures
- Do regression testing
- Evaluate bug sources…
EXAMPLE: QMAIL

Mail User Agent → Mail Transport Agent → Mail Delivery Agent → mbox

Mail User Agent ← Mail Transport Agent ← Mail Delivery Agent ← mbox

Mail User Agent → Mail Transport Agent → Mail Delivery Agent → mbox

Mail User Agent ← Mail Transport Agent ← Mail Delivery Agent ← mbox
QMAIL SECURITY

• Least privilege
  – Each module uses least privileges necessary
  – Only one setuid program
    • setuid to one of the other qmail user IDs, not root
    • No setuid root binaries
  – Only one process runs as root
    • Spawns the local delivery program under the UID and GID of the user being delivered to
    • No delivery to root
    • Always changes effective uid to recipient before running user-specified program

• Other secure coding ideas
QMAIL STRUCTURE

qmail-smtpd

Incoming SMTP mail

qmail-queue

Other incoming mail

qmail-send

qmail-rspawn

qmail-remote

qmail-lspawn

qmail-local
• **qmail-queue** runs as the special qmailq user.
• Manages the mail queue: qmail-queue is the only program in the system that can write to the queue.
• Signals qmail-send when the queue has messages
qmail-send runs as qmailq, signals qmail-lspawn on a local delivery, and qmail-remote if remote delivery
**QMAIL-LOCAL**

- **qmail-lspawn** runs as root. It looks at the message header, determines the user, and spawns **qmail-local**.
- **qmail-local** runs with ID of user receiving local mail.
qmail-local runs as the local mail recipient. It reads ~/.forward and decides whether to
- forward the message with qmail-queue, or
- append the message to the user’s inbox and exit.
**QMAIL-REMOTE**

`qmail-remote` runs as special user `qmailr`. It speaks SMTP and delivers messages to remote MTAs.
A program is isolated if it is unable to causally influence other programs on the system.

A sandbox is a mechanism for running a program so that it will be isolated.
WHY SANDBOX?

Opening untrusted files with executable content:

- MS Office documents
- Web pages
- Dancing hamster applications

More generally, to reinforce least privilege, modularity, and economy of mechanism
One approach is that taken by Qmail: install separate user accounts for each program.

What are the advantages of this approach?
Almost no programming required, uses “well-understood” tools...

What are the disadvantages?
Typical OS will allow shared resources “Default allow” policy
Some applications might not work
Processes that can’t make syscalls are isolated.
INTERPOSITION PROBLEMS

Race conditions, e.g.

1. open(/tmp/foo)
2. Check /tmp/foo, OK
3. /tmp/foo -> /not/allowed
4. open /tmp/foo
5. fd for /not/allowed
The sandbox needs to keep its own model of the state of the OS. This is called shadow state.
VIRTUAL OS

A sandbox with sufficient shadow state will end up “simulating” the whole OS.

E.g. sandbox keeps “cwd” variable, and translates system calls: open(x) to open(/cwd/x), etc.

To the application, the sandbox is the OS; To the OS, the sandbox and application look like any old process.
VIRTUAL MACHINES

Why not simulate the hardware as well? Then we have a **virtual machine**.

The VM simulates the processor, RAM, disk, etc. and updates the state on each instruction.

Since the application runs on a simulated rather than real machine, it can’t harm the physical machine, or other processes.
Virtual machines still leave the possibility of errors in the VM, or “covert” communication between VMs.

Processes can be isolated by running them on separate physical hardware.

Examples: Military SIPR Net, AV research, forensics