

UMSSIA

**LECTURE I:
SOFTWARE SECURITY**

THINKING LIKE AN ADVERSARY



SECURITY ASSESSMENT



Confidentiality?

Availability?

Dependability?

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

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 piece of memory.

BUFFER OVERFLOWS

Buffer overflows have been known since at least 1988 – the “Morris Worm” exploit.

Over 50% of CERT advisories in 2003 were buffer overflows. (15% of verified OSVDB reports in 2005)

If a C program has a buffer overflow, it can almost certainly be used for hijacking.

REVIEW

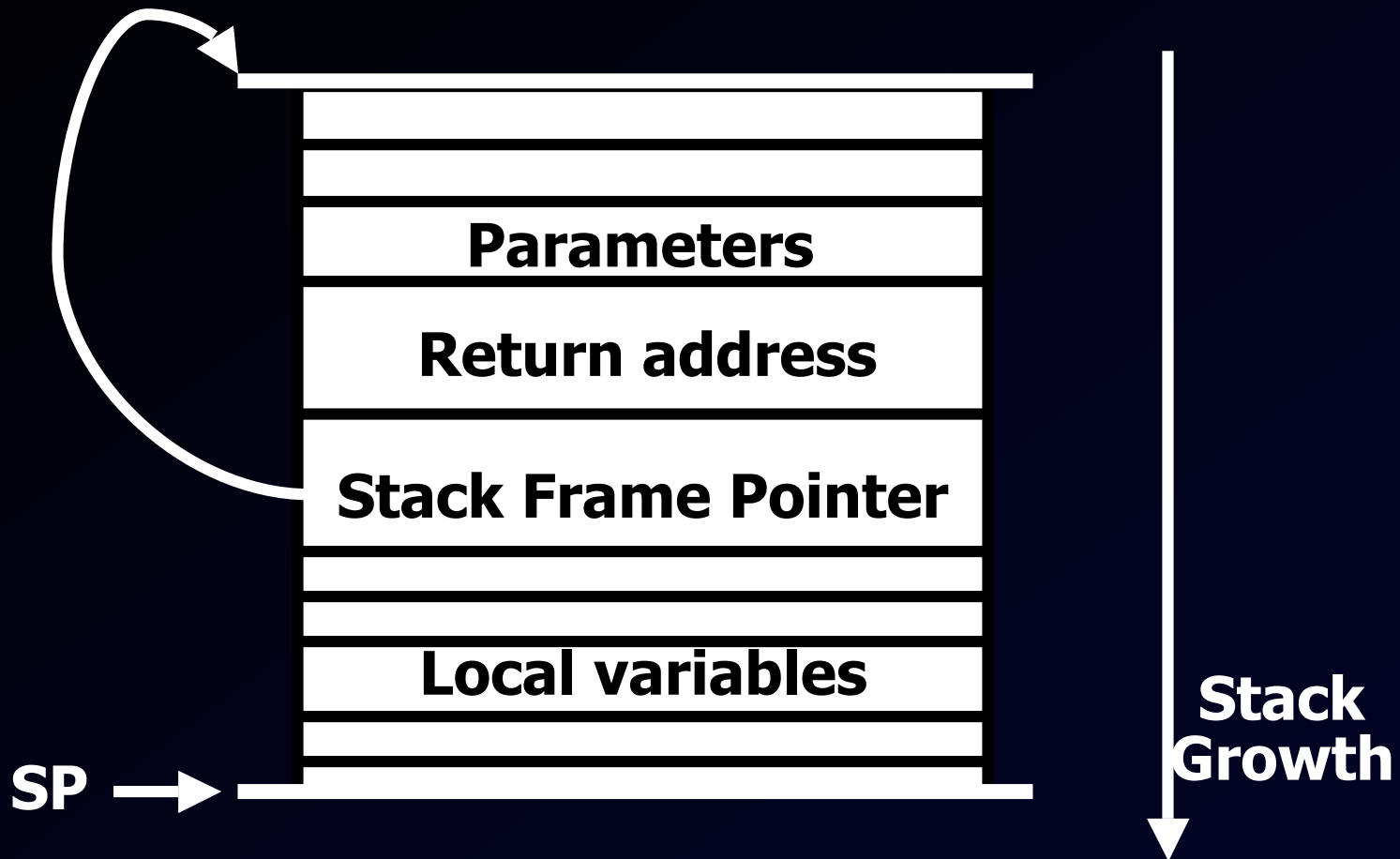
C functions store local variables on a “stack” including arguments and return address.

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

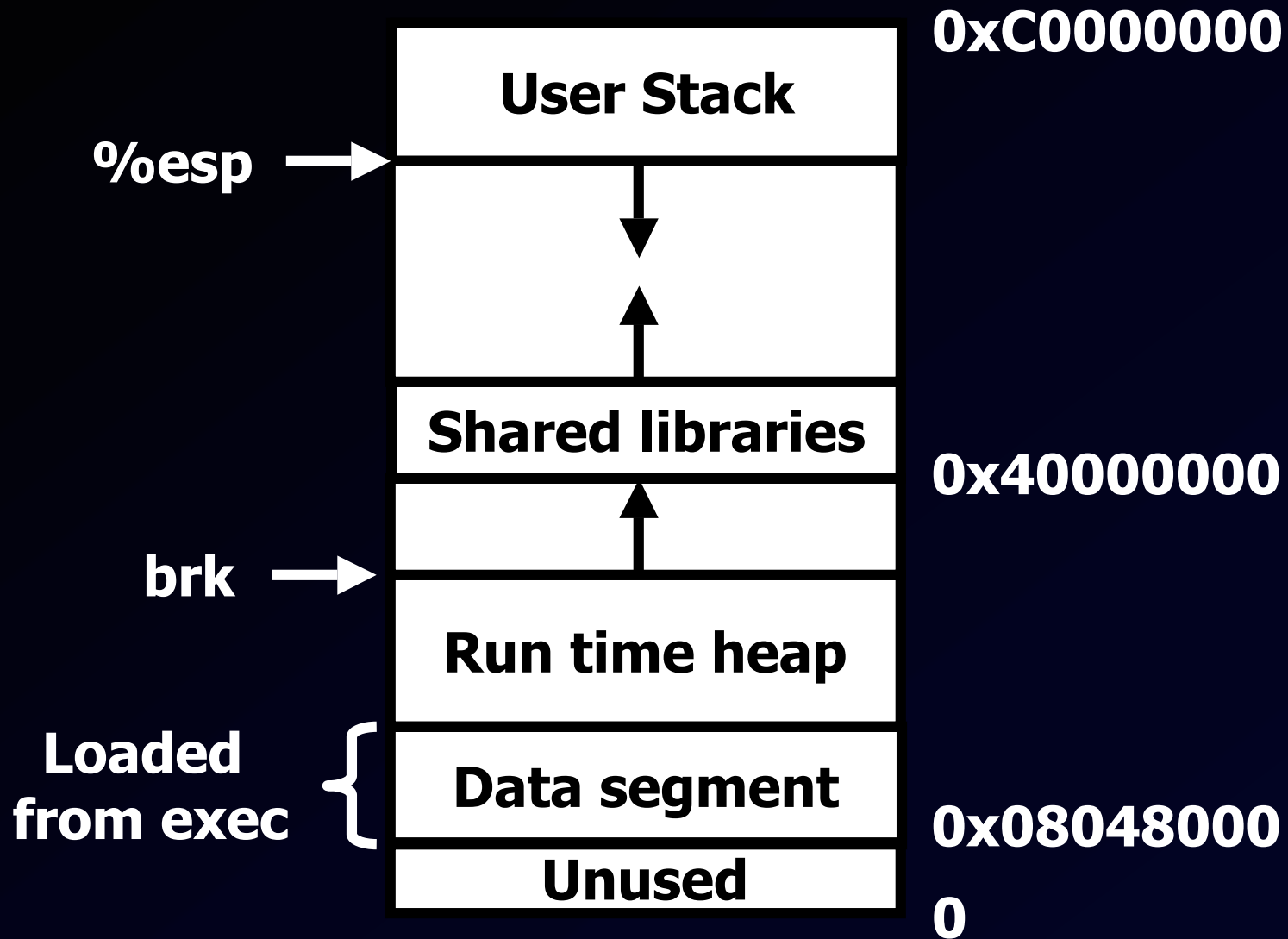
Running programs are stored in memory. The same code can have many stack frames.

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

THE STACK FRAME



LINUX MEMORY LAYOUT



BUFFER OVERFLOW IDEA

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

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

On enter:

(buf)	sfp	ret-addr	str
-------	-----	----------	-----

←

top of stack

After strcpy:

aa..aa	a..a	BCDE	X
--------	------	------	---

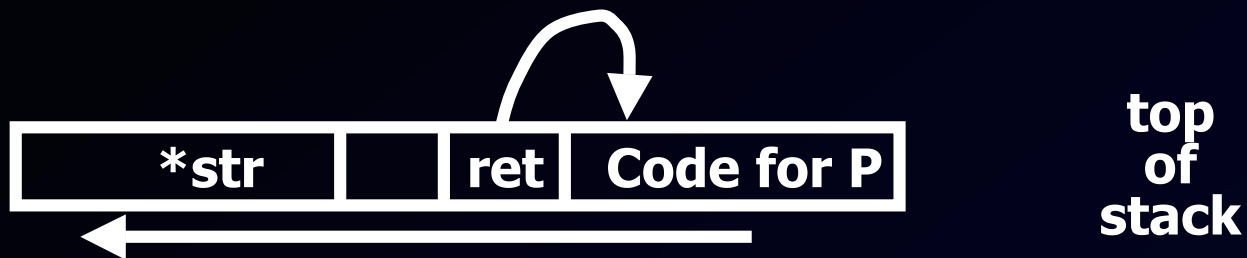
←

top of stack

On return: `sp = aaaa; jmp BCDE`

STACK EXPLOIT

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)

gets (char *s)

scanf (const char *format, ...)

sprintf (const char *format, ...)

⋮

HIJACKING METHODS

Control Flow can be hijacked in several ways:

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

<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>

Modifying **setjmp/longjmp buffers**.

<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.

PREVENTING OVERFLOWS

Buffer overflows happen because C lacks array Bounds. So **Don't Use C!**



Stuck with C?

Source Code Analysis

Non-executable stack

Randomization

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:



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!

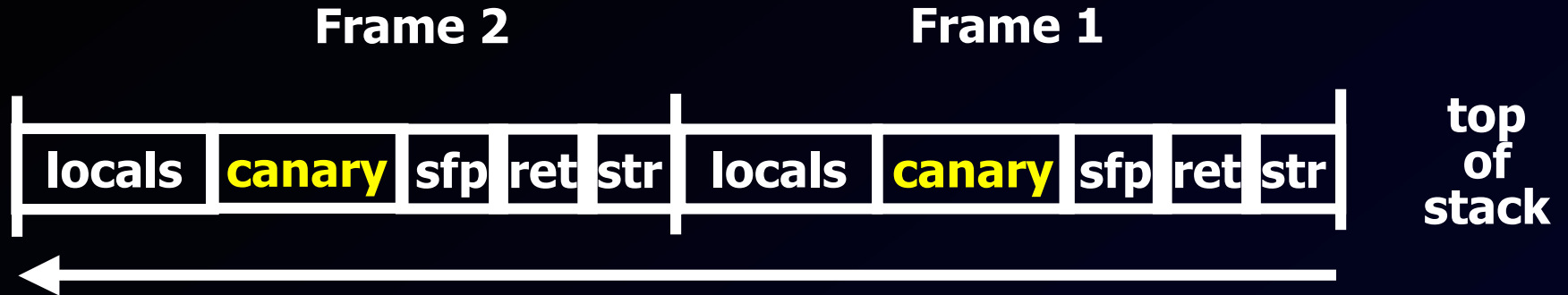
injected code usually won't run...

Both make exploits harder, but...

www.stanford.edu/~blp/papers/asrandom.pdf

www.usenix.org/events/sec05/tech/full_papers/sovarel/sovarel.pdf

RUN TIME CHECKING



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.

FORMAT STRING BUGS

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

Problem: what if user =
“%s%s%s%s%s%s%s”

Most likely program will crash...

If not, program will print memory contents.

Correct form:

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

EXPLOIT

Important things about format strings:

`printf("%4$c", 'a', 'b', 'c', 'd')` prints `'d'`.

`printf("%42x%n", 0, &var)` sets `var=42`.

Suppose we want to change the byte at address `ADDR` to `BB`, and know that the buffer with the format string is `WW-3` words up the stack:

`buf = "%BBx%WW$hhnxADDR"`

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"`

EXAMPLE

IIS has the **security goal** that only commands
In the subdirectory /scripts should be executed.

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

http://a.b.c.d/scripts/../../../../winnt/system32/cmd.exe?X

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**

INTEGER OVERFLOW

Machine integers are not real integers.

```
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 $a = 2^{32}-10$?

RACE CONDITIONS

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_XXXXXXXXX");  
fp = fopen(name, "w");
```

- Attacker creates symlink

/tmp/gs_12345A -> /etc/passwd

between call to mktmp and fopen, when root is running gs (just has to happen once!)

- Ghostscript (as root) overwrites /etc/passwd.

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**

V&M: SECURE THE WEAKEST LINK

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;  
}
```

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.**

ERROR CONDITIONS

- **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: $-1, 0, 1, 2^{31}-1, -2^{31} \dots$

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...**