Software Security: Dealing with C and C++

Dan Wallach

Rice University

Problems with C and C++

- No memory safety / type safety guarantees
  - Cast pointers to integers
  - No bounds checking on arrays
  - Unitialized contents from `malloc()`
  - Reuse memory after `free()`
- Results?

```
Segmentation fault
Core dumped
```

But we need C and C++

- Huge installed base of software / libraries
- Supports every possible platform
- Mature development tools

- Security issues?
  - Buffer overflow attacks
  - Malformed input → crashes
  - Excessive trust of input (SQL injection, etc.)

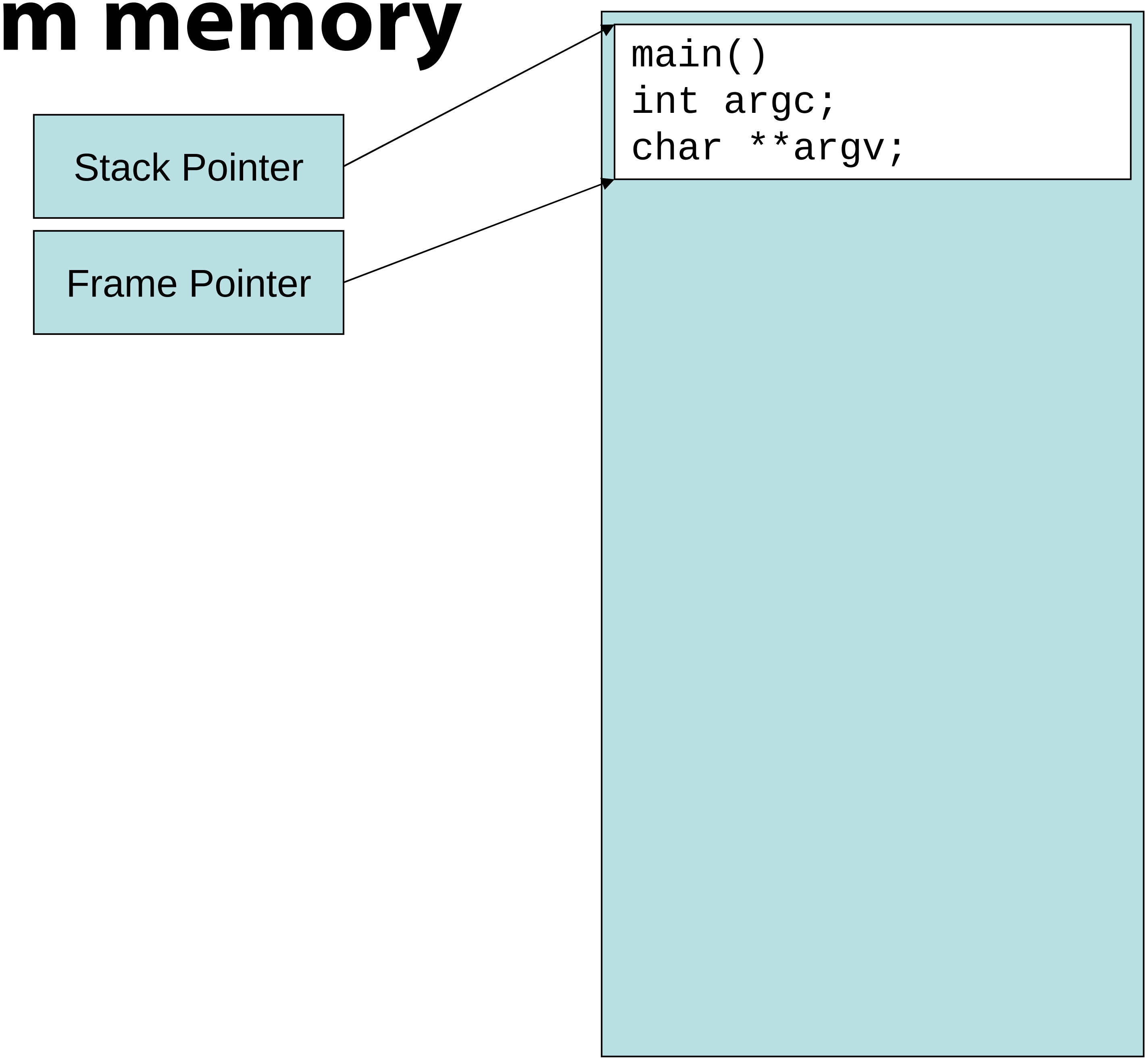
Anatomy of a buffer overflow

```
void LogText(char *message) {
  char buf[MAXBUF];

  sprintf(buf, “%s
    getdate(), messag

  ...
}
```

M XBUF is huge.
lo sweat.

Attacker: what if message is larger than MAXBUF?

System memory

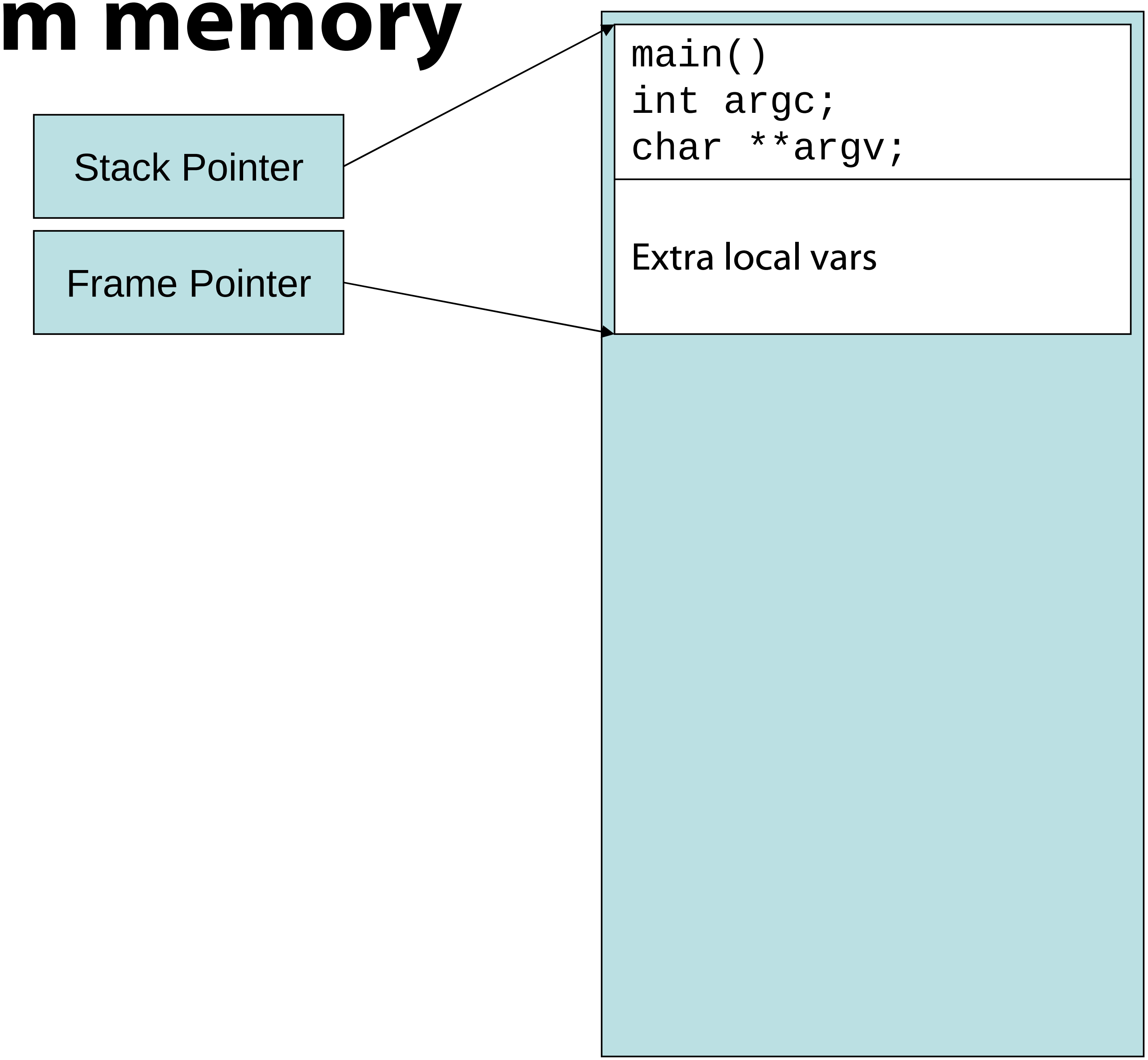


Stack Pointer

Frame Pointer

```
main()
int argc;
char **argv;
```

System memory



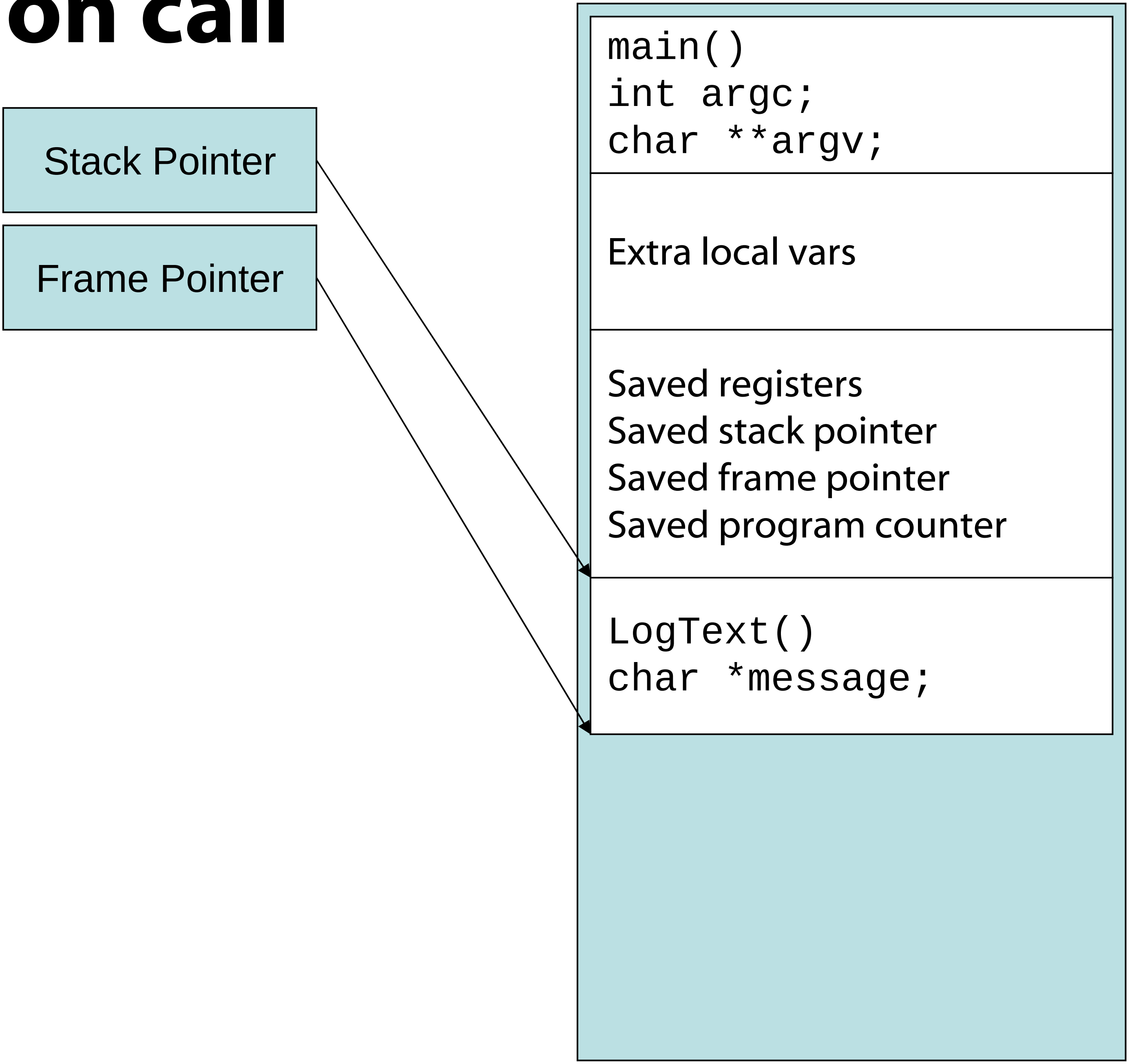
Stack Pointer

Frame Pointer

```
main()
int argc;
char **argv;
```

Extra local vars

Function call



Stack Pointer

Frame Pointer

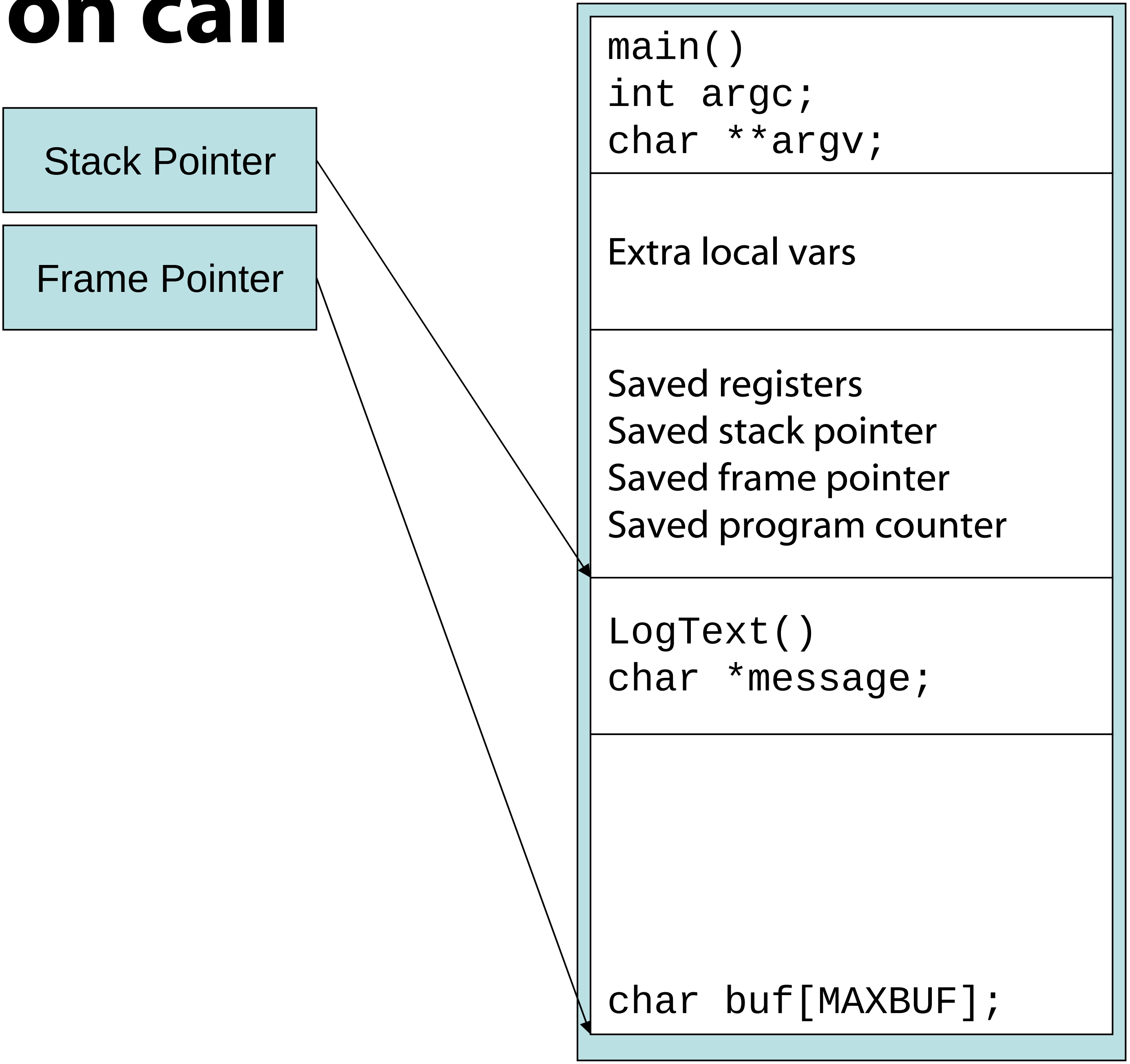
```
main()
int argc;
char **argv;
```

Extra local vars

Saved registers
Saved stack pointer
Saved frame pointer
Saved program counter

```
LogText()
char *message;
```

Function call



Stack Pointer

Frame Pointer

```
main()
int argc;
char **argv;
```

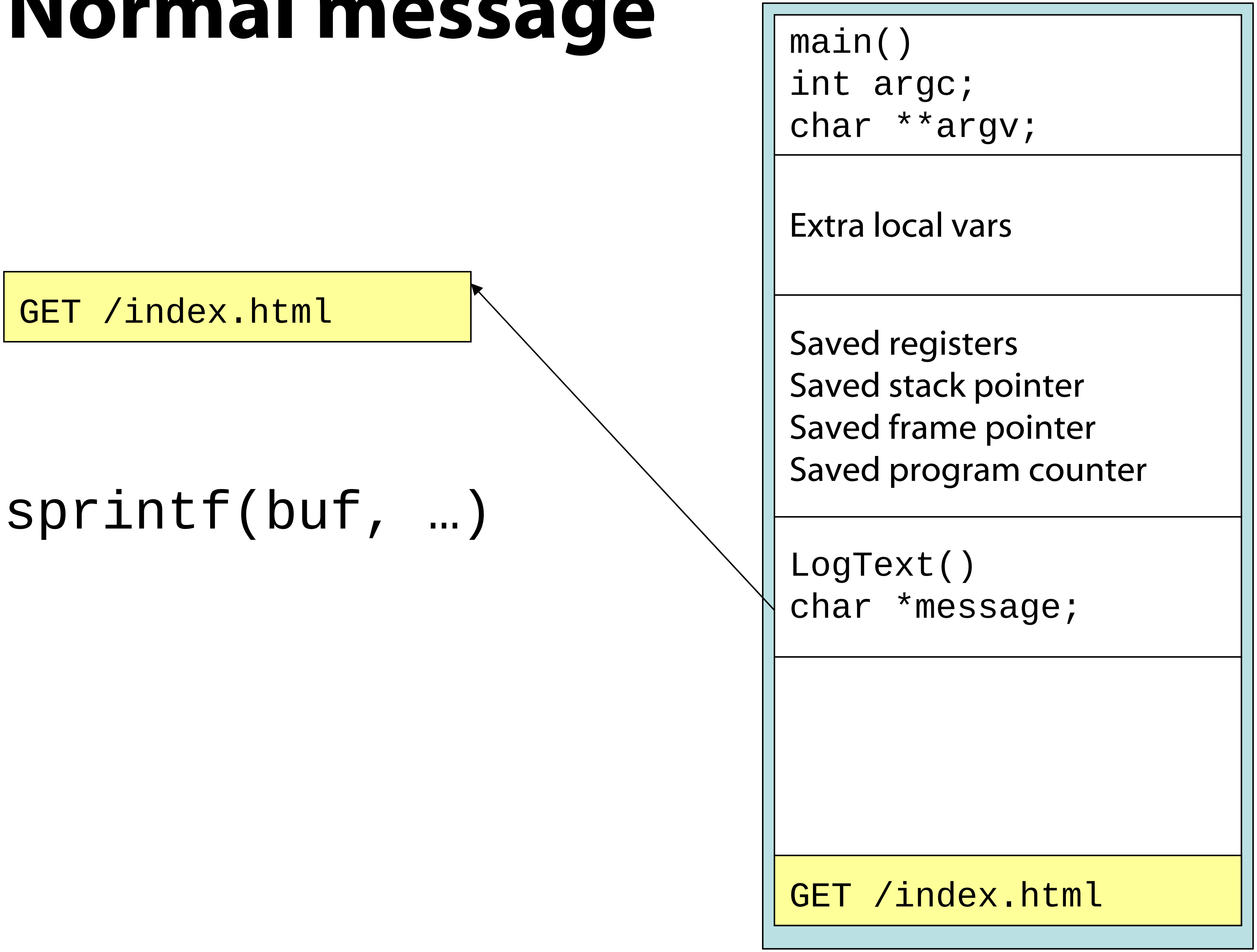
Extra local vars

Saved registers
Saved stack pointer
Saved frame pointer
Saved program counter

```
LogText()
char *message;
```

```
char buf[MAXBUF];
```

Normal message



```
GET /index.html
```

```
sprintf(buf, …)
```

```
main()
int argc;
char **argv;
```

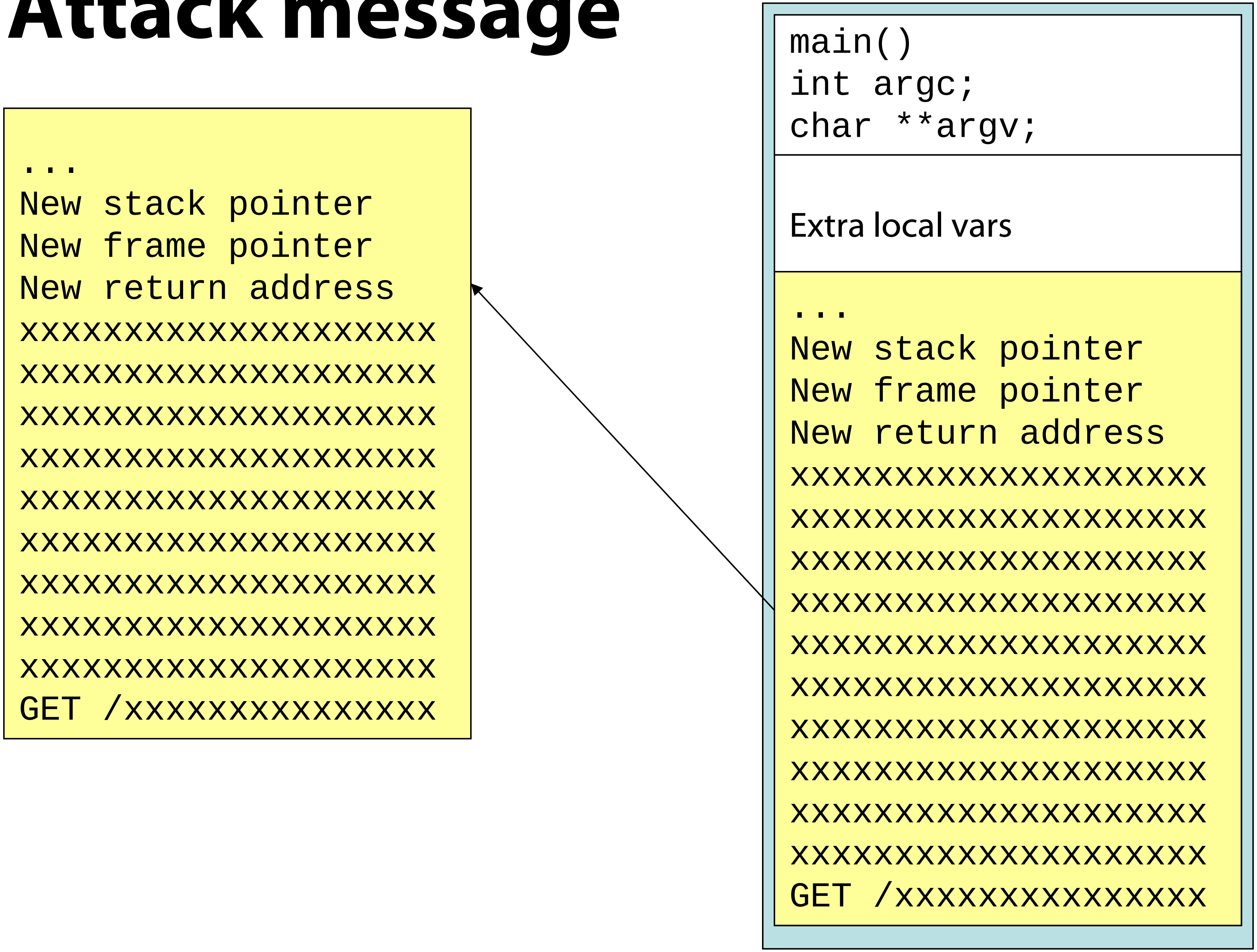
Extra local vars

Saved registers
Saved stack pointer
Saved frame pointer
Saved program counter

```
LogText()
char *message;
```

```
GET /index.html
```

Attack message



```
...
New stack pointer
New frame pointer
New return address
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
GET /xxxxxxxxxxxxxxx
```

```
main()
int argc;
char **argv;
```

Extra local vars

```
...
New stack pointer
New frame pointer
New return address
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
GET /xxxxxxxxxxxxxxx
```

Buffer overflows

- Overwrite return address
  - Option #1: call into your own buffer
  - Option #2: set up a stack frame, call elsewhere `system(“cat /etc/passwd | mail…”)`

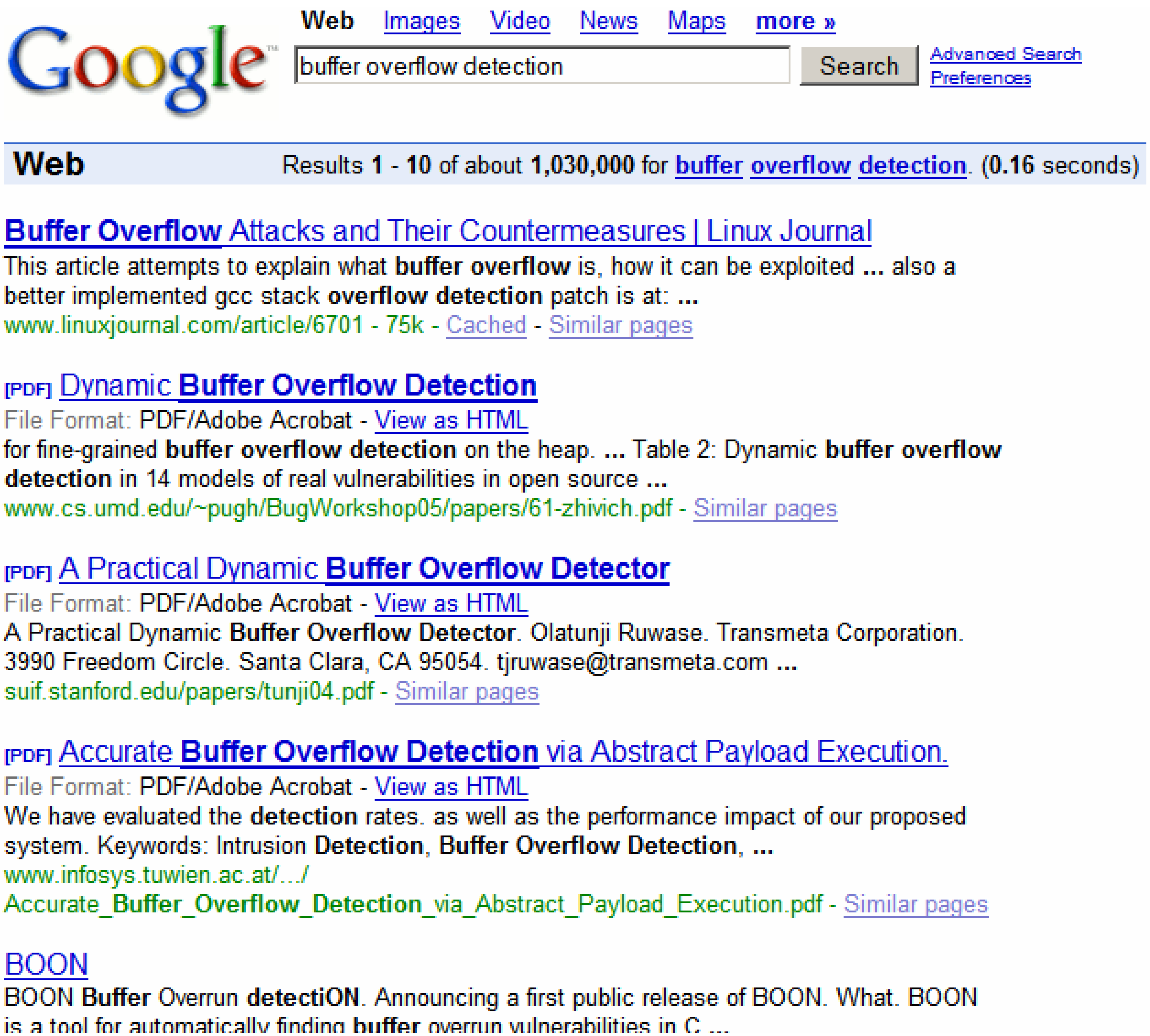
- Attackers don’t need source code
  - Plenty of attacks on Windows
  - Generate garbage input, inspect crash dumps (“fuzzing”)

Solution: good string hygiene

- *Never* use `sprintf()`, `gets()`, `strcpy()` or other functions that don't know buffer sizes
- Instead, see `snprintf()` or `asprintf(), strncpy(), ...`

- But what if you forget something?

Lots and lots of solutions…



buffer overflow detection

Web Results **1** - **10** of about **1,030,000** for **buffer overflow detection**. (**0.16** seconds)

Buffer Overflow Attacks and Their Countermeasures | Linux Journal
This article attempts to explain what **buffer overflow** is, how it can be exploited **...** also a better implemented gcc stack **overflow detection** patch is at: **...**
www.linuxjournal.com/article/6701 - 75k - Cached - Similar pages

[PDF] Dynamic **Buffer Overflow Detection**
File Format: PDF/Adobe Acrobat - View as HTML
for fine-grained **buffer overflow detection** on the heap. **...** Table 2: Dynamic **buffer overflow detection** in 14 models of real vulnerabilities in open source **...**
www.cs.umd.edu/~pugh/BugWorkshop05/papers/61-zhivich.pdf - Similar pages

[PDF] A Practical Dynamic **Buffer Overflow Detector**
File Format: PDF/Adobe Acrobat - View as HTML
A Practical Dynamic **Buffer Overflow Detector**. Olatunji Ruwase. Transmeta Corporation. 3990 Freedom Circle. Santa Clara, CA 95054. tjruwase@transmeta.com **...**
suif.stanford.edu/papers/tunji04.pdf - Similar pages

[PDF] Accurate **Buffer Overflow Detection** via Abstract Payload Execution.
File Format: PDF/Adobe Acrobat - View as HTML
We have evaluated the **detection** rates. as well as the performance impact of our proposed system. Keywords: Intrusion **Detection**, **Buffer Overflow Detection**, **...**
www.infosys.tuwien.ac.at/.../
Accurate_**Buffer**_**Overflow**_**Detection**_via_Abstract_Payload_Execution.pdf - Similar pages

BOON
BOON **Buffer** Overrun **detectiON**. Announcing a first public release of BOON. What. BOON is a tool for automatically finding **buffer** overrun vulnerabilities in C ...

This lecture

- Runtime solutions (e.g., StackGuard)
- Compile-time static analysis
- Software engineering for security

Runtime solutions

Started with StackGuard [Cowan et al., 1998]

- “Canaries” surround the return value
- Validate the canaries before returning

Standard feature on modern C++ compilers

- gcc 4.1 has `–fstack-protector`
- MS Visual Studio 7.0 has `/GS` flag

- Modest performance cost
  - Enabled by default in OpenBSD

StackGuard discussion

- Defeats code injection and *return-to-libc* attacks

- No protection against *heap overflows*
- Cannot patch pre-compiled binaries
- More subtle attacks may still work (e.g., modify a code pointer on the stack)
  - In C++, lots of code pointers around

No eXecute page bits

- Recent x86 architectural feature
  (existed on many other CPUs for years)
- Code pages must be marked executable
- Executable pages are not writable
- Stack is not executable

- Eliminates attacks that inject code
- Does not prevent return-to-libc attacks
- Some programs may break

Other approaches

- Grow the stack up instead of down
  - Doesn’t work so well on x86
- Address space randomization
  - Change locations of libraries / functions
  - Works well with a sparse 64-bit address space
  - Brute force attacks possible with 32-bit addrs

- Use a better programming language
  - More on this later…

Static analysis

- Growing industry (Coverity, Fortify, ...)
- Many open source tools
  - C/C++: BOON, MOPS, CQual, splint, ...
  - Java: ESC/Java2, FindBugs
- Complete program coverage
  - Tools will follow obscure code paths
- Non-trivial programmer overhead
  - Annotating code to help the scanner
  - Studying output, dealing with false positives

Example: user/kernel data analysis

- CQual uses data flow analysis
  - Can identify use of “tainted” data in an untainted context
- Reading user data in Linux kernel
  - Proper behavior: Copy data from user to kernel space with safe routine, then parse
  - Annotations: label user pointers on the way in, forbid dereferencing

Other analyses

- Untrusted (network) data never used …
  - as `printf` format string
  - as part of an SQL command
  - as part of HTML output (cross-site scripting)
- Incorrect `malloc` / `free` behavior
- Y2K bugs
- Device drivers following rules

Microsoft device driver dev tools

PRE*f*ast For Drivers (PFD)

- Lightweight and fast (runs in minutes)
- Easy to use early in development – start early
  - Use on any code that compiles
- Limited to a procedure scope
- Works on any code, C and C++
- Finds many local violations

Static Driver Verifier (SDV)

- Extremely deep analysis (runs in hours)
- More useful in the later stages of development
  - Requires complete driver
  - Works over the whole driver
- Limited to WDM and to C (more planned)
- Finds deep bugs

Static analysis summary

- Powerful tools now available (open and commercial)
- Excellent at finding obscure bugs
- Still an area of active research

Intrusions happen

- What do you do *after* an intrusion?
  - Restore from backups?
  - Identify / block attack route?

- How do you *detect* an intrusion?

- What if the intrusion compromises the whole operating system? (Rootkits)

Intrusion detection systems

- Host-based (system call tracing)
    - Antivirus software
- Network-based (packet sniffing)
    - Email scanners
    - Firewalls

- Large industry + lots of open software

The value of honeypots

- Honeypot: a machine/service expecting no legitimate traffic
  - No worries about false positives
  - Any activity is intruder activity
- Save everything (useful for forensics)

- State of the art: zero-day attack detection
  - Detect new attacks fast
  - Propagate attack signatures quickly

Why not just use a safe language?

coverity

- **Checks include:**
  - Buffer overflows
  - Cross-site scripting
  - Denial of service
  - File corruption
  - Format string vulnerabilities
  - Improper bounds checking
  - Insecure access control
  - Integer overflows
  - Memory corruption
  - Out-of-bounds array access
  - Privilege escalations
  - SQL injection

Java

- **Remaining issues:**
  -
  - Cross-site scripting
  - Denial of service
  -
  -
  -
  - Insecure access control
  -
  -
  -
  - Privilege escalations
  - SQL injection

Architecting security

It’s not about the programming language

Basic principles, best designed from the start

- Always check your input
- Separation / modularity
- Least privilege
- Threat modeling / analysis
- Software engineering processes

Don’t trust your input

- A huge source of real-world problems
  - SQL injection attacks
  - Cross-site scripting attacks
  - Format string / buffer overflow attacks
- Don’t even trust “trusted” input
  - Configuration files

➔Easiest change you can retrofit to an existing system.

Digresion #1: Avoid mobile code

- Temptation: use general-purpose PL interpreter as file format
  - Postscript vs. PDF
- If necessary, remove dangerous primitives

  Microsoft print driver, rasterizing example:
  - No need for file access
    - Limited font loading functionality
  - No need for network access

Separation / modularity

- Fault containment
  - Watchdog processes, etc.
- Narrow interfaces
  - Avoid fragile class hierarchies
  - Easier to replace / re-engineer components
- Wrappers on legacy software?

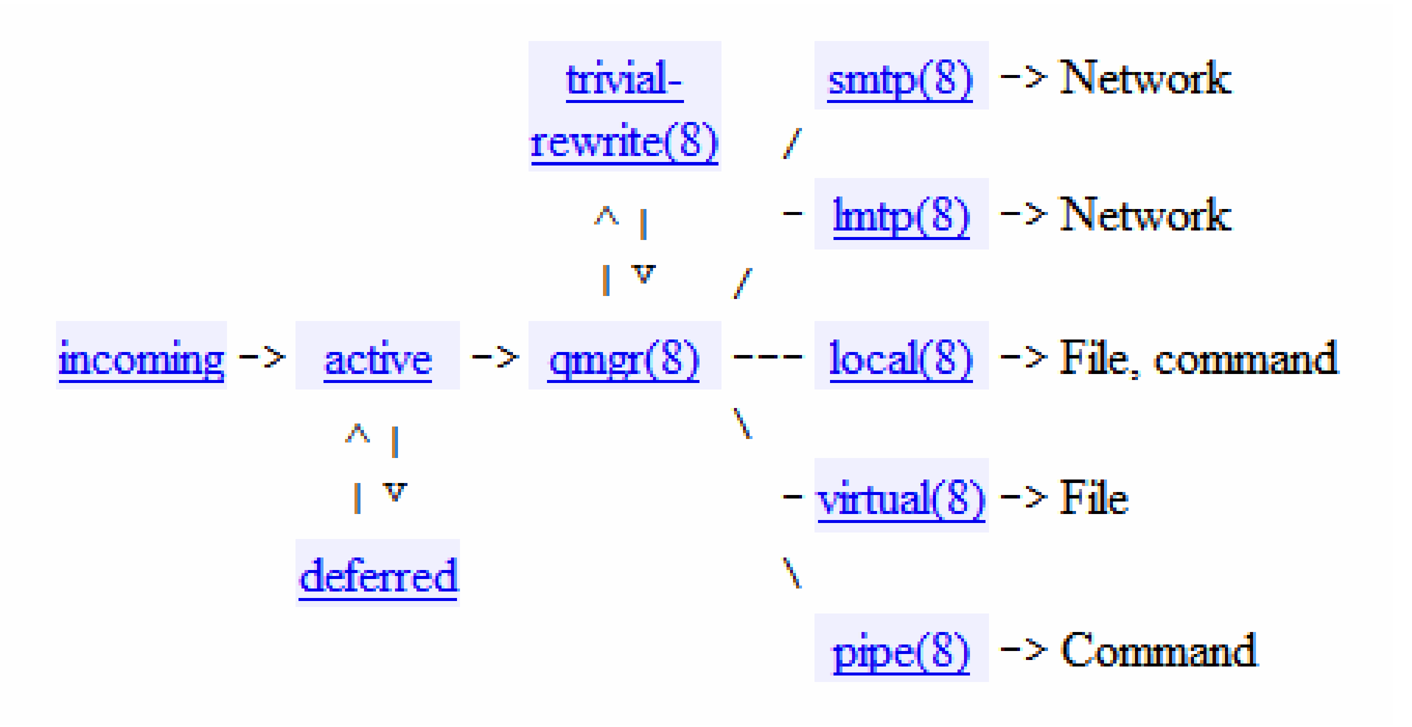
Least privilege

- Most valuable idea in software architecture
- Different modules need different privileges

- Reduce the size of trusted components
  - Less code to audit for correctness
  - Limit damage from a security compromise

Least privilege with OS features

- Separate user ids for different programs
  - Limited privileges for most users
- Example: *postfix* mail transport agent



Digression #2: setuid, chroot

- Temptation: run as root, emulate user
  1. stat() file owner / permissions
  2. Read/write as superuser
- Risk: attacker may replace file

  (Time of check to time of use attack)
- Preferable: setuid() to the user
- Related: use chroot() rather than parsing filenames to restrict a directory

Threat modeling

What’s going to go wrong?

- Hardware failure
- Software corner-case bugs
- Flash crowds (“Slashdot effect”)

Adversaries

- Theft of service (rootkits / zombies)
- Read / leak secrets (credit card numbers)
- Write / modify data
- Insider threats?

Plan in advance!

Software engineering process

Any process is better than no process.

- Software version control
- Unit testing
- Code reviews
- Pair programming
- Rapid prototyping

Any good idea can be overdone.

- Design patterns

Duff’s Law

“Whenever possible, steal code.”

- Somebody else maintains it
  - Example: OpenSSL, rapid security fixes
- Avoid making subtle mistakes
  - Notable problem with crypto & network protocols
- More time on your own code

Example: Banks / e-Commerce

Hardware failure

- Time is money; aggressive replication

Obscure bugs

- Load testing with real traces
- "Fuzz" testing (random inputs)

Flash crowds

- Over-provision + estimates of worst-case
- Service prioritization?

Bank adversaries

Theft of service

- Aggressive / annoying firewalls & IDS
- Human monitoring
- Regularly reinstall computers from scratch

Read / write secrets (i.e., steal money)

- “Red Team” (adversarial) code analysis
- Online auditing / redundant records

Insider threats

- Separation of user privileges

What about…

Aircraft control software?

- No malicious users / developers
- Higher reliability requirements

Consumer operating system?

- Uses / configurations you can’t anticipate
- Importance of crash recovery

Voting machine software?

- Every person (developers, poll workers, voters) may be malicious!

(More on voting machines, later)

Upcoming lectures

- Java architectures for safety / security
  - Least privilege with PL mechanisms
- Distributing your system over a network
  - Using structured p2p overlays