Memory safety, continued

With material from Mike Hicks and Dave Levin

Last time

- More memory attacks
- Started: Principled defense: Making memory violations impossible
Today

- More on principled defenses, then
- Avoiding exploitation
  - Memory violations possible but not harmful
A memory safe program execution:

1. Only creates pointers through standard means
   - \( p = \text{malloc}(...) \), or \( p = \&x \), or \( p = \&\text{buf}[5] \), etc.

2. Only uses a pointer to access memory that “belongs” to that pointer

Combines two ideas:

temporal safety and spatial safety
Spatial safety

- View pointers as **capabilities**: triples \((p, b, e)\)
  - \(p\) is the actual pointer (current address)
  - \(b\) is the base of the memory region it may access
  - \(e\) is the extent (bounds) of that region (count)

- **Access allowed** iff \(b \leq p \leq (e - \text{sizeof} \text{(typeof}(p)))\)
Memory safety for C

• C/C++ are here to stay.
  • You can write memory safe programs with them
  • But the language provides no guarantee

• Compilers could add code to check for violations
  • Out-of-bounds: immediate failure (Java ArrayBoundsException)

• This idea has been around for more than 20 years. Performance has been the limiting factor.
  • Work by Jones and Kelly in 1997 adds 12x overhead
  • Valgrind memcheck adds 17x overhead
Research progress

- **CCured** (2004), 1.5x slowdown
  - But no checking in libraries
  - Compiler rejects many safe programs

- **Softbound/CETS** (2010): 2.16x slowdown
  - Complete checking, highly flexible

- **Intel MPX** hardware (2015 in Linux)
  - Hardware support to make checking faster

Type Safety
Type safety

• Each object is ascribed a type (int, pointer to int, pointer to function), and

• Operations on the object are always compatible with the object’s type
  • Type safe programs do not “go wrong” at run-time

• **Type safety** is **stronger** than memory safety

```c
int (*cmp)(char*, char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*, char*))p;
cmp("hello", "bye"); // crash!
```

Memory safe, NOT type safe
Aside: Dynamic Typing

- Dynamically typed languages
  - Don’t require type declaration
  - e.g., Ruby and Python
  - Can be viewed as type safe

- Each object has one type: **Dynamic**
  - Each operation on a Dynamic object is permitted, but *may be unimplemented*
  - In this case, it *throws an exception*
  - Checked at **runtime** not **compile time**!
Enforce invariants

• Types most useful for enforcing invariants
  • (Examples next slide)

• Enforcement of abstract types: modules with hidden representation
  • Allow reasoning more confidently about their isolation from the rest of the program

For more on type safety, see http://www.pl-enthusiast.net/2014/08/05/type-safety/
Types for Security

- Use types to enforce **security property** invariants
  - Invariants about data’s privacy and integrity
  - Enforced by the type checker

- **Example**: *Java with Information Flow (JIF)*

```java
int{Alice, Bob} x;
int{Alice, Bob, Chuck} y;
x = y; // OK: policy on x is stronger
y = x; // BAD: policy on y is weaker
```

http://www.cs.cornell.edu/jif
Why not type safety?

• C/C++ often chosen for performance reasons
  • Manual memory management
  • Tight control over object layouts
  • Interaction with low-level hardware

• Enforcement of type safety is typically expensive
  • Garbage collection avoids temporal violations
    • Can be as fast as malloc/free, often uses much more memory
  • Bounds and null-pointer checks avoid spatial violations
  • Hiding representation may inhibit optimization
    • Many C-style casts, pointer arithmetic, & operator, not allowed
A new hope?

• Many applications do not need C/C++
  • Or the risks that come with it

• New languages aim to provide similar features to C/C++ while remaining type safe
  • Google’s Go, Mozilla’s Rust, Apple’s Swift
Avoiding exploitation
Until we have a widespread type-safe replacement for C, what can we do?

- Make bugs **harder to exploit**
  - Crash but not code execution

- **Avoid bugs** with better programming
  - Secure coding practices, code review, testing

**Better together**: Try to avoid bugs, *but also* add protection if some slip through
Avoiding exploitation

Recall the steps of a stack smashing attack:

• Putting attacker code into memory
  • (No zeroes or other stoppers)

• Getting %eip to point to an address you specify

• Finding the correct address

How can we make these attack steps more difficult?
• Side note: How to implement fixes?

• Goal: change libraries, compiler, or OS
  • Fix *architectural design*, not code
  • Avoid changing (lots of) application code
  • One update fixes all programs at once
Avoiding exploitation

Recall the steps of a stack smashing attack:

• Putting attacker code into memory
  • (No zeroes or other stoppers)
• Getting %eip to point to address you specify
• Finding the correct address

How can we make these attack steps more difficult?
Detecting overflows with **canaries**

19th century coal mine integrity
- Is the mine safe?
- Dunno; bring in a canary
- If it dies, abort!

*We can do the same for stack integrity!*
Detecting overflows with **canaries**

Not the expected value: abort!

What value should the canary have?

Check canary just before every function return.
Canary values

From StackGuard (your reading)

1. Terminator canaries (CR, LF, NUL (i.e., 0), -1)
   - Leverages the fact that scanf etc. don’t allow these

2. Random canaries
   - Write a new random value @ each process start
   - Save the real value somewhere in memory
   - Must write-protect the stored value

3. Random XOR canaries
   - Same as random canaries
   - But store canary XOR some control info, instead
Other canary tricks

- Put canaries in heap metadata
- Reorganize locals to put buffers above pointers
  - Buffers can only overwrite themselves, canary
  - [ProPolice]
- Global return stack [StackShield]
  - Copy ret address from separate stack every time
Canary weaknesses

- Overwrite function pointer
- Overwrite local variable pointer to indirectly reference eip
- Anything not stack (heap, etc.)
- Bad randomization
- Memory is not necessarily secret
  - Buffer overreads
  - Read via crash
Overread example

From Strackx et al.

```c
void vulnerable(char *name_in) {
    char buf[10];
    strncpy(buf, name_in, sizeof(buf));
    printf("Hello, \%s\n", buf);
}
```

- Strncpy is “safe” because it won’t overwrite
- But string not properly terminated

name_in = “0123456789ABC”

buf does not append NULL

prints until NULL

Text: 02 8d e2 10 %ebp %eip &arg1

buf canary
Avoiding exploitation

Recall the steps of a stack smashing attack:

- Putting attacker code into memory
  **Defense: Stack Canaries**
- Getting `eip` to point to an address you specify
- Finding the correct address

How can we make these attack steps more difficult?
• Goal: Don’t run attacker code

• Defense: Make stack non-executable

• Try to jump to attacker shellcode in the stack, panic instead

http://www.ipadforums.net/wallpapers/data/2/DontPanic.png
Return-to-libc

%eip

buffer

padding

known location

Only need to know where libc is

nop sled malicious code

libc

exec()

printf()

"/bin/sh"

libc

libc

libc

libc
Avoiding exploitation

Recall the steps of a stack smashing attack:

- Putting attacker code into memory
  
  **Defense: Stack Canaries**

- Getting `%eip` to point to address you specify
  
  **Defense: Non-executable stack (kind of)**

- Finding the correct address

How can we make these attack steps more difficult?
Address-space layout randomization

- Randomly place some elements in memory
- Make it hard to find libC functions
- Make it hard to guess where stack (shellcode) is
Return-to-libc, thwarted

%eip

padding

unknown locations

Text

libc

buffer

exec()

printf()

"/bin/sh"

libc
ASLR today

• Available on modern operating systems
  • Linux in 2004, other systems slowly afterwards; most by 2011

• Caveats:
  • **Only shifts the offset** of memory areas
    • Not locations within those areas
    • Possible to use a read exploit to find it
  • **May not apply to program code**, just libraries
  • **Need sufficient randomness**, or can brute force
    • 32-bit systems: typically 16 bits = 65536 possible starting positions; sometimes 20 bits. Shacham brute force attack could defeat this in 216 seconds (2004 hardware)
    • 64-bit systems more promising, e.g., 40 bits possible
Cat and mouse

- **Defense**: Make stack/heap non-executable to prevent injection of code
  - **Attack response**: Return to libc

- **Defense**: Hide the address of desired libc code or return address using ASLR
  - **Attack response**: Brute force search or information leak

- **Defense**: Avoid using libc code entirely and use code in the program text instead
  - **Attack response**: Construct needed functionality using return oriented programming (ROP)
Return oriented programming (ROP)
Return-oriented Programming

• Introduced by Hovav Shacham, CCS 2007

• Idea: rather than use a single (libc) function to run your shellcode, string together pieces of existing code, called gadgets, to do it instead

• Challenges
  • **Find the gadgets** you need
    • String them together
Approach

• Gadgets are instruction groups that end with `ret`

• Stack serves as the code
  • `%esp` = program counter
  • Gadgets invoked via `ret` instruction
  • Gadgets get their arguments via `pop`, etc.
    • Also on the stack
Simple example

0x17f: pop %edx
ret

goal: put 5 into edx
mov %edx, 5

“Instructions”

%esp
%eip

Gadget

“program counter”

%edx 5

Text ... 0x17f 5 next

0x00 0xffffffff
Code sequence (no ROP)

0x17f: mov %eax, [%esp]
mov %ebx, [%esp+8]
mov [%ebx], %eax

%eax | 5
%ebx | 0x404

Text  5  ⋯  5  ⋯  0x404  ⋯  ⋯
Equivalent ROP sequence
Return-Oriented Programming is a lot like a ransom note, but instead of cutting out letters from magazines, you are cutting out instructions from text segments.
Whence the gadgets?

• How can we find gadgets to construct an exploit?
  • Automated search: look for ret instructions, work backwards
    • Cf. https://github.com/0vercl0k/rp

• Are there sufficient gadgets to do anything interesting?
  • For significant codebases (e.g., libc), Turing complete
    • Especially true on x86’s dense instruction set
  • Schwartz et al. (USENIX Sec’11) automated gadget shellcode creation, Turing complete not required
Blind ROP

• **Defense**: Randomizing the location of the code (by compiling for position independence) on a 64-bit machine makes attacks very difficult
  • Recent, published attacks are often for 32-bit versions of executables

• **Attack response**: Blind ROP

• If server restarts on a crash, but does not re-randomize:
  1. Read the stack to **leak canaries and a return address**
  2. Find a few gadgets (at run-time) to **effect call to write**
  3. Dump binary to find gadgets for shellcode

http://www.scs.stanford.edu/brop/
Blind ROP, continued

- Able to completely automatically, only through remote interactions, develop a remote code exploit for nginx, a popular web server
  - The exploit was carried out on a 64-bit executable with full stack canaries and randomization

- Conclusion: Are avoidance defenses hopeless?
  - Put another way: Memory safety is really useful!
Today

- Finish up memory safety:
  - Finish CFI
  - Rules for secure coding in C
- Move on to malware
  - Viruses
  - Worms
  - Case studies
  - “Modern” malware
Control Flow
Integrity
Behavior-based detection

- Stack canaries, non-executable data, ASLR make standard attacks harder / more complicated, but may not stop them

- Idea: **observe** the program’s **behavior** — is it doing **what we expect it to**?
  - If not, might be compromised

- Challenges
  - Define “expected behavior”
  - Detect deviations from expectation efficiently
  - Avoid compromise of the detector
Control-flow Integrity (CFI)

- Define “expected behavior”:
  - Control flow graph (CFG)

- Detect deviations from expectation efficiently

- Avoid compromise of the detector
Call Graph

sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x<y;
}
bool gt(int x, int y) {
    return x>y;
}

Which functions call other functions
sort2(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x<y;
}
bool gt(int x, int y) {
    return x>y;
}

Break into basic blocks
Distinguish calls from returns
CFI: Compliance with CFG

- **Compute the call/return CFG** in advance
  - During compilation, or from the binary

- **Monitor the control flow** of the program and ensure that it only follows paths allowed by the CFG

- Observation: **Direct calls** need not be monitored
  - Assuming the code is immutable, the target address cannot be changed

- Therefore: **monitor only indirect calls**
  - jmp, call, ret with non-constant targets
Control Flow Graph

```c
bool lt(int x, int y) {
    return x<y;
}

bool gt(int x, int y) {
    return x>y;
}

sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}
```

**Direct calls** *(always the same target)*
Control Flow Graph

sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x<y;
}

bool gt(int x, int y) {
    return x>y;
}

**Indirect transfer** *(call via register, or ret)*
Control-flow Integrity (CFI)

• Define “expected behavior”:
  Control flow graph (CFG)

• Detect deviations from expectation efficiently
  In-line reference monitor (IRM)

• Avoid compromise of the detector
In-line Monitor

- Implement the monitor in-line, as a program transformation

- Insert a label just before the target address of an indirect transfer

- Insert code to check the label of the target at each indirect transfer
  - Abort if the label does not match

- The labels are determined by the CFG
Memory safety, continued, continued
Control-flow Integrity (CFI)

- Define “expected behavior”: **Control flow graph** (CFG)

- Detect deviations from expectation efficiently: **In-line reference monitor** (IRM)

- Avoid compromise of the detector
Use the same label at all targets:
label just means it’s OK to jump here.

What could go wrong?
Recall

Simplest labeling

• Can’t return to functions that aren’t in the graph

• **Can** return to the right function in the wrong order
Detailed labeling

- All potential destinations of **same source** must match
  - Return sites from calls to **sort** must share a label (**L**)
  - Call targets **gt** and **lt** must share a label (**M**)
  - Remaining label unconstrained (**N**)

Prevents more abuse than simple labels, **but still permits call from site A to return to site B**
Classic CFI instrumentation

Before CFI

<table>
<thead>
<tr>
<th>FF 53 08</th>
<th>call [ebx+8] ; call a function pointer</th>
</tr>
</thead>
</table>

is instrumented using `prefetchnta` destination IDs, to become:

<table>
<thead>
<tr>
<th>8B 43 08</th>
<th>mov eax, [ebx+8] ; load pointer into register</th>
</tr>
</thead>
<tbody>
<tr>
<td>3E 81 78 04 78 56 34 12</td>
<td>cmp [eax+4], 12345678h ; compare opcodes at destination</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label ; if not ID value, then fail</td>
</tr>
<tr>
<td>FF D0</td>
<td>call eax ; call function pointer</td>
</tr>
<tr>
<td>3E 0F 18 05 DD CC BB AA</td>
<td>prefetchnta [AABBCDDdh] ; label ID, used upon the return</td>
</tr>
</tbody>
</table>

After CFI

Fig. 4. Our CFI implementation of a call through a function pointer.

<table>
<thead>
<tr>
<th>Bytes (opcodes)</th>
<th>x86 assembly code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 10 00</td>
<td>ret 10h</td>
<td>; return, and pop 16 extra bytes</td>
</tr>
</tbody>
</table>

is instrumented using `prefetchnta` destination IDs, to become:

<table>
<thead>
<tr>
<th>8B 0C 24</th>
<th>mov ecx, [esp] ; load address into register</th>
</tr>
</thead>
<tbody>
<tr>
<td>83 C4 14</td>
<td>add esp, 14h ; pop 20 bytes off the stack</td>
</tr>
<tr>
<td>3E 81 79 04 DD CC BB AA</td>
<td>cmp [ecx+4], AABBCDDdh ; compare opcodes at destination</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label ; if not ID value, then fail</td>
</tr>
<tr>
<td>FF E1</td>
<td>jmp ecx ; jump to return address</td>
</tr>
</tbody>
</table>
Classic CFI instrumentation

```
FF 53 08  call [ebx+8] ; call a function pointer

is instrumented using prefetchnta destination IDs, to become:

8B 43 08  mov  eax, [ebx+8] ; load pointer into register
3E 81 78 04 78 56 34 12  cmp  [eax+4], 12345678h ; compare opcodes at destination
75 13  jne  error_label ; if not ID value, then fail
FF D0  call eax ; call function pointer
3E 0F 18 05  prefetchnta [AABBCDDh] ; label ID, used upon the return

Fig. 4. Our CFI implementation of a call through a function pointer.
```

<table>
<thead>
<tr>
<th>Bytes (opcodes)</th>
<th>x86 assembly code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 10 00</td>
<td>ret 10h           ; return, and pop 16 extra bytes</td>
<td></td>
</tr>
</tbody>
</table>

```
is instrumented using prefetchnta destination IDs, to become:

8B 0C 24  mov  ecx, [esp] ; load address into register
83 C4 14  add  esp, 14h ; pop 20 bytes off the stack
3E 81 79 04 DD CC BB AA  cmp  [ecx+4], AABBCDDh ; compare opcodes at destination
75 13  jne  error_label ; if not ID value, then fail
FF E1  jmp  ecx ; jump to return address
```
Efficient?

- **Classic CFI** (2005) imposes *16% overhead* on average, *45%* in the **worst case**
  - Works on arbitrary executables
  - Not modular (no dynamically linked libraries)

- **Modular CFI** (2014) imposes *5% overhead* on average, *12%* in the **worst case**
  - C only (part of LLVM)
  - Modular, with separate compilation
  - [http://www.cse.lehigh.edu/~gtan/projects/upro/](http://www.cse.lehigh.edu/~gtan/projects/upro/)
Control-flow Integrity (CFI)

- Define “expected behavior”:
  Control flow graph (CFG)

- Detect deviations from expectation efficiently
  In-line reference monitor (IRM)

- Avoid compromise of the detector
  Sufficient randomness, immutability
Can we defeat CFI?

• **Inject code** that has a **legal label**
  • *Won’t work* because we assume **non-executable data**

• **Modify code labels** to allow the desired control flow
  • *Won’t work* because the **code is immutable**

• **Modify stack during a check**, to make it seem to succeed
  • *Won’t work* because **adversary cannot change registers** into which we load relevant data
  • No time-of-check, time-of-use bug (TOCTOU)
CFI Assurances

• CFI defeats **control flow-modifying** attacks
  • Remote code injection, ROP/return-to-libc, etc.

• But **not manipulation of control-flow** that is **allowed by the labels/graph**
  • Called **mimicry attacks**
  • The simple, single-label CFG is susceptible to these

• **Nor data leaks or corruptions**
  • Heartbleed would not be prevented
  • Nor the **authenticated** overflow
    • Which is allowed by the graph

```c
void func(char *arg1) {
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, str);
    if(authenticated) { …
    }
```
Secure?

• MCFI can **eliminate 95.75% of ROP gadgets** on x86-64 versions of SPEC2006 benchmark suite
  • By ruling their use non-compliant with the CFG

• Average Indirect-target Reduction (AIR) > **99%**
  • Essentially, the percentage of **possible targets of indirect jumps** that CFI rules out
Secure Coding
Secure coding in C

- Since the language provides few guarantees, developers must use discipline.

- Good reference guide: CERT C Coding Standard
  - [https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard](https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard)
  - Similar guides for other languages (e.g., Java)
  - See also: David Wheeler: [http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/internals.html](http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/internals.html)

Combine with advanced code review and testing
Design vs. Implementation

• In general, we strive to follow principles and rules
  • A principle is a design goal with many possible manifestations.
  • A rule is a specific practice consistent with sound principles.
    • The difference between these can sometimes be fuzzy

• Here we look at rules for good C coding
  • In particular, to avoid implementation errors that could result in violations of memory safety

• Later: Consider principles and rules more broadly
General Principle: Robust coding

• Like defensive driving
  • Avoid depending on anyone else around you
  • If someone does something unexpected, you won’t crash (or worse)
  • It’s about minimizing trust

• Each module pessimistically checks its assumed preconditions (on outside callers)
  • Even if you “know” clients will not send a NULL pointer
  • … Better to throw an exception (or even exit) than run malicious code
int main() {
    char buf[100], *p;

    while (1) {
        p = fgets(buf,sizeof(buf),stdin);
        len = atoi(p);
        p = fgets(buf,sizeof(buf),stdin);
        len = MIN(len,strlen(buf));
        for (i=0; i<len; i++) {
            if (!iscntrl(buf[i]))
                putchar(buf[i]);
            else putchar('.');
        }
        printf("\n");
    }
...
Rule: Enforce input compliance

```c
char digit_to_char(int i) {
    char convert[] = "0123456789";
    if(i < 0 || i > 9)
        return '?';
}
```

- **Unfounded trust** in received input is a recurring source of vulnerabilities
- We will see many more examples in the course
Rule: Use safe string functions

- Traditional string library routines assume target buffers have sufficient length
  
  ```c
  char str[4];
  char buf[10] = "good";
  strcpy(str,"hello"); // overflows str
  strcat(buf,"day to you"); // overflows buf
  ```

- Safe versions check the destination length
  
  ```c
  char str[4];
  char buf[10] = "good";
  strlcpy(str,"hello",sizeof(str)); // fails
  strlcat(buf,"day to you",sizeof(buf)); // fails
  ```
Detour: strncpy vs. strlcpy

Recall:

```c
void vulnerable(char *name_in) {
    char buf[10];
    strncpy(buf, name_in, sizeof(buf));
    printf("Hello, %s\n", buf);
}
```

- `strncpy` is “safe” because it won’t overwrite
  - But string not properly terminated
  - Always add `buf[sizeof(buf) - 1] = 0;`
- `strlcpy` is better — copies (n-1) bytes max and appends the null for you!
Replacements

• ... for string-oriented functions
  • `strcat` ➔ `strlcat`
  • `strcpy` ➔ `strlcpy`
  • `strncat` ➔ `strlcat`
  • `strncpy` ➔ `strlcpy`
  • `sprintf` ➔ `snprintf`
  • `vssprintf` ➔ `vsnprintf`
  • `gets` ➔ `fgets`

• Microsoft versions different
  • `strcpy_s`, `strcat_s`, ...
Rule: Don’t forget NUL terminator

- Strings require one additional character to store the NUL. Forgetting that could lead to overflows.

```c
char str[3];
strcpy(str,"bye"); // write overflow
int x = strlen(str); // read overflow
```

- Using safe string library calls will catch this mistake

```c
char str[3];
strlcpy(str,"bye",3); // stops safely
int x = strlen(str); // returns 2
```
Rule: Understand pointer arithmetic

- `sizeof()` returns number of bytes, but pointer arithmetic multiplies by the `sizeof` the type

```c
int buf[SIZE] = { ... };
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
    *buf_ptr++ = getnext(); // will overflow
}
```

- So, use the right units

```c
while (!done() && buf_ptr < (buf + SIZE)) {
    *buf_ptr++ = getnext(); // stays in bounds
}
```
Principle: Defend dangling pointers

```c
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p’s space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```
Rule: Use NULL after free

```c
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; // defend against bad deref
int **q = malloc(sizeof(int*)); // reuses p’s space
*q = &x;
*p = 5; // (good) crash
**q = 3;
```
**Principle:** Manage memory properly

```c
int foo(int arg1, int arg2) {
    struct foo *pf1, *pf2;
    int retc = -1;

    pf1 = malloc(sizeof(struct foo));
    if (!isok(arg1)) goto DONE;
    ...
    pf2 = malloc(sizeof(struct foo));
    if (!isok(arg2)) goto FAIL_ARG2;
    ...
    retc = 0;

    FAIL_ARG2:
    free(pf2); //fallthru
DONE:
    free(pf1);
    return retc;
}
```

- **Rule:** Use goto chains to avoid duplicated or missed code
  - Mimics try/finally in languages like Java
- **Confirm your logic!**
  - *Gotofail bug*
Anatomy of a `goto fail`

```c
static OSStatus
SSLVerifySignedServerKeyExchange(...)
{
    OSStatus err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    goto fail; // triggers if if fails: err == 0
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...
    // SSL verify called somewhere in here

    fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err; // returns err = 0 (SUCCESS), without SSL verify function
}
```
Rule: Favor safe libraries

- Designed to ensure safe use of strings, pointers, etc.
  - Encapsulate well-thought-out design. Take advantage!

- **Smart pointers**
  - Pointers with only safe operations
  - Lifetimes managed appropriately
  - First in the Boost library, now a C++11 standard

- **Networking**: Google protocol buffers, Apache Thrift
  - For dealing with network-transmitted data
  - Ensures input validation, parsing, etc.
  - Efficient
Rule: Use a safe allocator

• ASLR challenges libc exploits by making the library base unpredictable

• **Challenge heap-based overflows** by making the **addresses** returned by `malloc` **unpredictable**
  • Can have some negative performance impact

• Example implementations:
  • Windows Fault-Tolerant Heap
  • **DieHard** (on which fault-tolerant heap is based)
    • [http://plasma.cs.umass.edu/emery/diehard.html](http://plasma.cs.umass.edu/emery/diehard.html)