0x1A Great Papers in Computer Security

Vitaly Shmatikov

http://www.cs.utexas.edu/~shmat/courses/cs380s/

Course Logistics

- Lectures: Tuesday and Thursday, 2-3:15pm
- Instructor: Vitaly Shmatikov
  - Office: CSA 1.114
  - Office hours: Tuesday, 3:30-4:30pm (after class)
  - Open door policy – don't hesitate to stop by!
- TA: Martin Georgiev
  - Office hours: Wednesday 1:30-3pm, PAI 5.33
- No textbook; we will read a fair number of research papers
- Watch the course website for lecture notes, assignments, and reference materials

Grading

- Homeworks: 40% (4 homeworks, 10% each)
  - Homework problems will be based on research papers
- Midterm: 15%
- Project: 45%
  - Computer security is a contact sport – the best way to understand it is to get your hands dirty
  - Projects can be done individually or in small teams
  - Project proposal due September 20
  - You can find a list of potential project ideas on the course website, but don't hesitate to propose your own

Prerequisites

- Basic understanding of operating systems and memory management
  - At the level of an undergraduate OS course
- Some familiarity with cryptography is helpful
  - Cryptographic hash functions, public-key and symmetric cryptosystems
- Undergraduate course in complexity and/or theory of computation
- Ask if you are not sure whether you are qualified to take this course

What This Course is Not About

- Not a comprehensive course on computer security
- Not a course on cryptography
  - We will cover some crypto when talking about cryptographic protocols and privacy
- Not a seminar course
  - We will read and understand state-of-the-art research papers, but you'll also have to do some actual work ☺
- Focus on several specific research areas
  - Mixture of theory and systems (very unusual!)
- You have a lot of leeway in picking your project

"Best Hits" Course

- 26 selected papers
  - Somewhat arbitrary – a reflection of personal taste
  - Complete list on the website
  - Will also discuss follow-up and related work
- Goal: give you a taste of what research in computer security is like
- Wide variety of topics
  - Memory attacks and defenses, secure information flow, understanding Internet-wide worms and viruses, designing and breaking cryptographic protocols, anonymity and privacy, side-channel attacks…

Start Thinking About a Project

- A few ideas are on the course website
- Many ways to go about it
  - Build a tool that improves software security
    - Analysis, verification, attack detection, attack containment
  - Apply an existing tool to a real-world system
  - Demonstrate feasibility of some attack
  - Do a substantial theoretical study
  - Invent something of your own
- Start forming teams and thinking about potential topics early on!

A Few Project Ideas

- Security of cloud computing (Amazon EC2, etc.)
- Errors in security logic of Web applications
- Unintended leakages and covert channels
- Anonymous communication schemes
- Privacy issues in networked consumer devices
- Security of Android APIs
- Wireless routing, authentication, localization
- Security for voice-over-IP
- Choose something that interests you!

C. Cowan, P. Wagle, C. Pu, S. Beattie, J. Walpole

Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade

(DISCEX 1999)

Famous Internet Worms

- Morris worm (1988): overflow in fingerd
  - 6,000 machines infected (10% of existing Internet)
- CodeRed (2001): overflow in MS-IIS server
  - 300,000 machines infected in 14 hours
- SQL Slammer (2003): overflow in MS-SQL server
  - 75,000 machines infected in **10 minutes** (!!)
- Sasser (2004): overflow in Windows LSASS
  - Around 500,000 machines infected

Responsible for user authentication in Windows

… And The Band Marches On

- Conficker (2008-09): overflow in Windows RPC
  - Around 10 million machines infected (estimates vary)
- Stuxnet (2009-10): several zero-day overflows + same Windows RPC overflow as Conficker
  - Windows print spooler service
    - Also exploited by Flame (announced in 2012)
  - Windows LNK shortcut display
  - Windows task scheduler

Why Are We Insecure?

[Chen et al. 2005]

- 126 CERT security advisories (2000-2004)
- Of these, 87 are memory corruption vulnerabilities
- 73 are in applications providing remote services
  - 13 in HTTP servers, 7 in database services, 6 in remote login services, 4 in mail services, 3 in FTP services
- Most exploits involve illegitimate control transfers
  - Jumps to injected attack code, return-to-libc, etc.
  - Therefore, most defenses focus on control-flow security
- But exploits can also target configurations, user data and decision-making values

Memory Exploits

- Buffer is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
  - If executable code is supplied as "data", victim's machine may be fooled into executing it
    - Code will self-propagate or give attacker control over machine
- Attack can exploit any memory operation
  - Pointer assignment, format strings, memory allocation and de-allocation, function pointers, calls to library routines via offset tables
  - Attacks need not involve injected code!

Stack Buffers

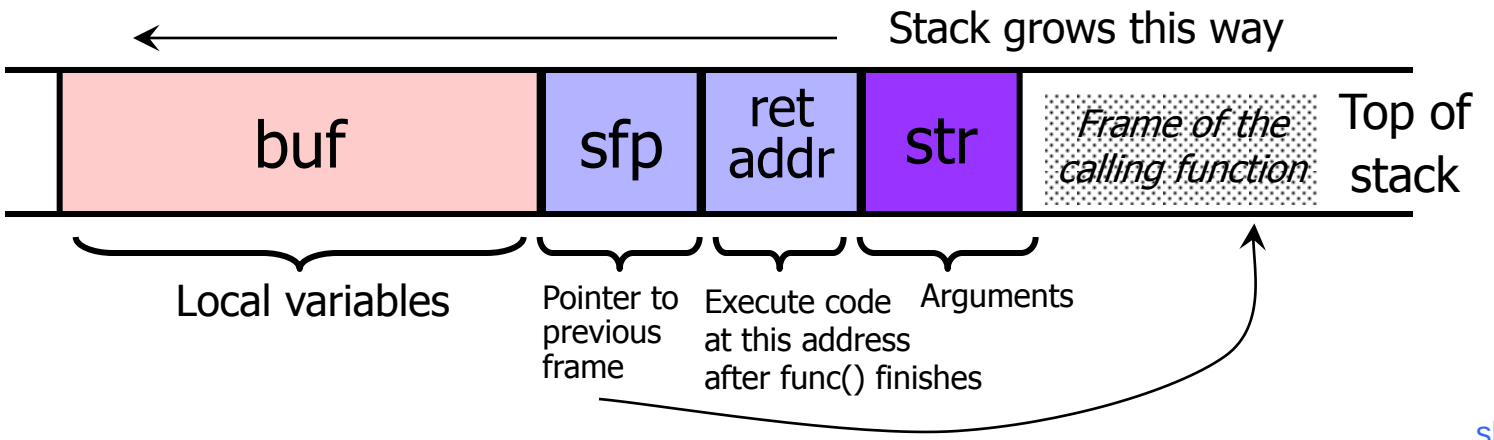
- Suppose Web server contains this function

```
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

Allocate local buffer
(126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame (activation record) is pushed onto the stack



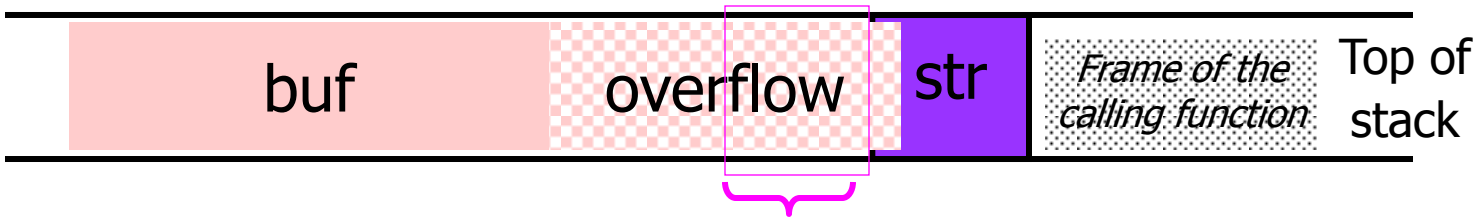
What If Buffer Is Overstuffed?

- Memory pointed to by str is copied onto stack...

```
void func(char *str) {
     char buf[126];
     strcpy(buf,str);
}
```

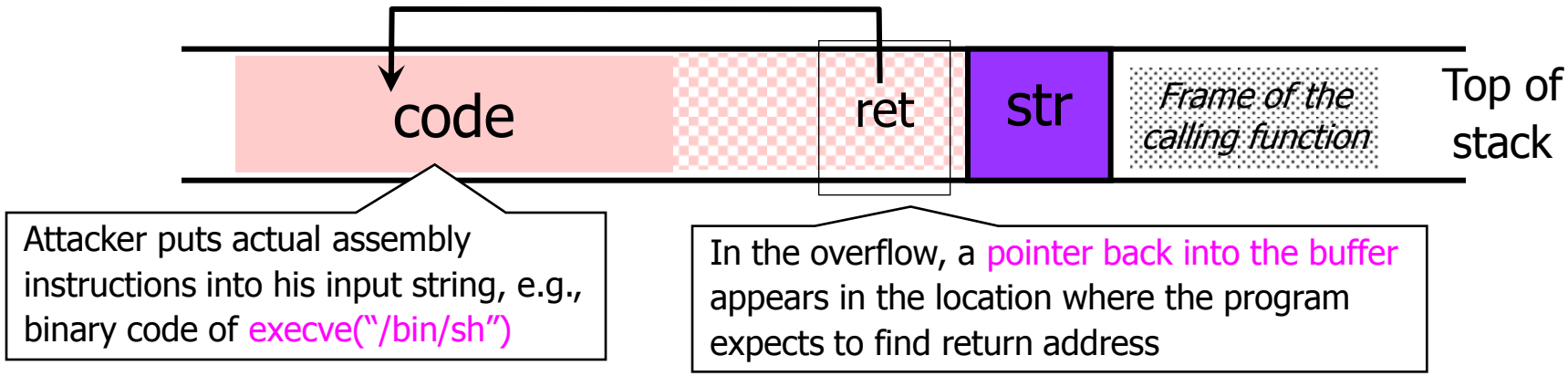
strcpy does NOT check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations



Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, str points to a string received from the network as the URL



- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

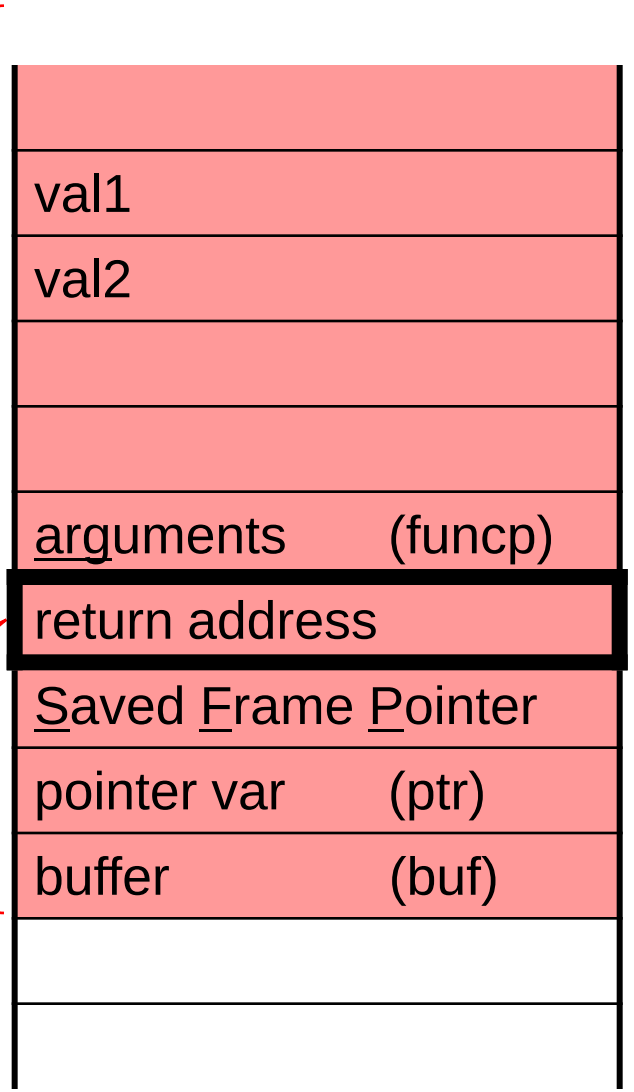
Basic Stack Code Injection

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but…
- For the basic stack-smashing attack, overflow portion of the buffer must contain correct address of attack code in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
    - Otherwise application will crash with segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called

Stack Corruption: General View

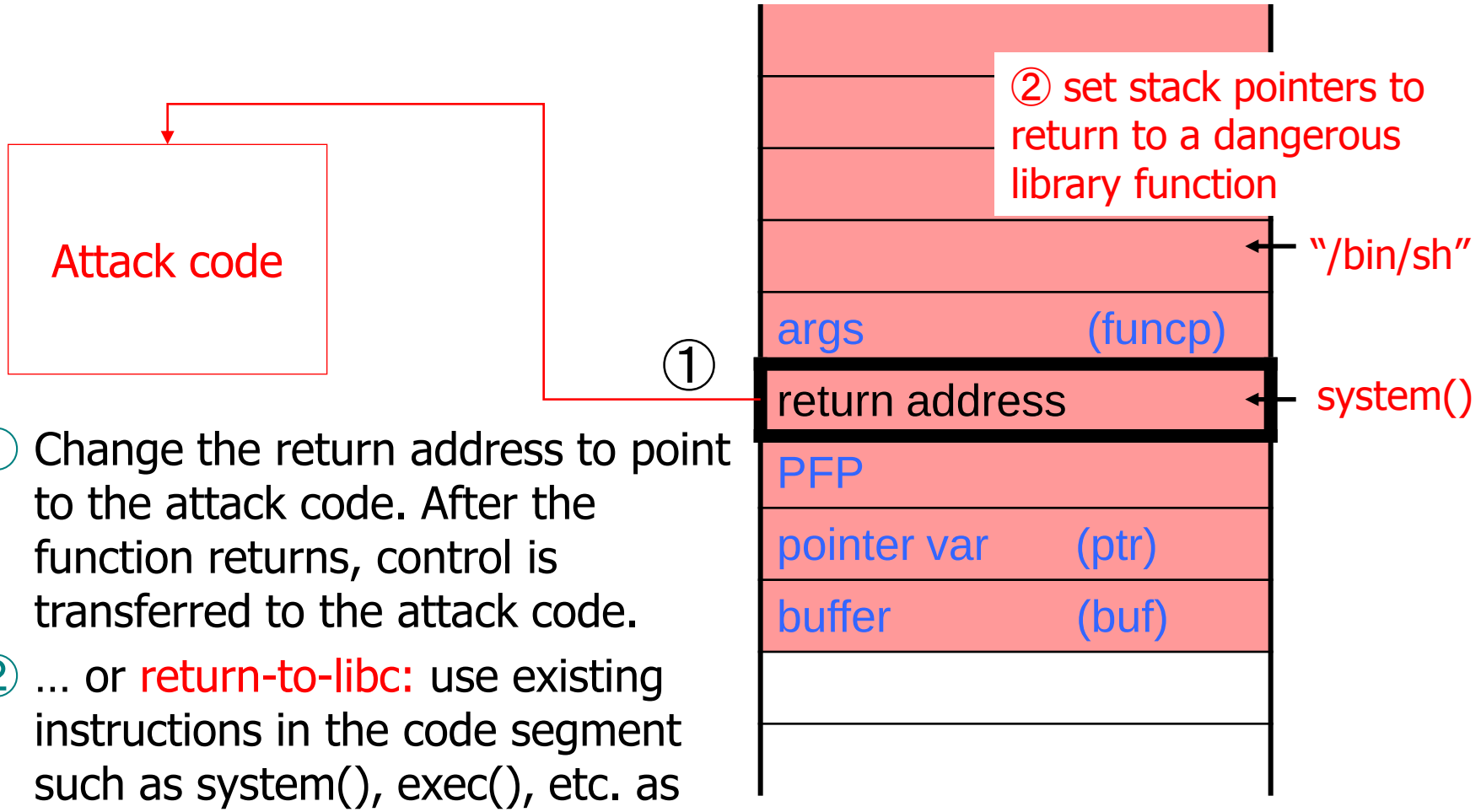
```
int bar (int val1) {
    int  val2;
    foo (a_function_pointer);
}

int foo (void (*funcp)()) {
    char* ptr = point_to_an_array;
    char buf[128];
    gets (buf);
    strncpy(ptr, buf, 8);
    (*funcp)();
}
```

Attacker-controlled memory



Most popular target

Attack #1: Return Address



① Change the return address to point to the attack code. After the function returns, control is transferred to the attack code.

② … or return-to-libc: use existing instructions in the code segment such as system(), exec(), etc. as the attack code.

Cause: No Range Checking

- strcpy does not check input size
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until "\0" is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, ...)
  - printf(const char *format, ...)

Does Range Checking Help?

- strncpy(char *dest, const char *src, size_t n)
  - If strncpy is used instead of strcpy, no more than n characters will be copied from *src to *dest
  - Programmer has to supply the right value of n
- Potential overflow in htpasswd.c (Apache 1.3)

```
… strcpy(record,user);
  strcat(record,":");
  strcat(record,cpw); …
```

Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

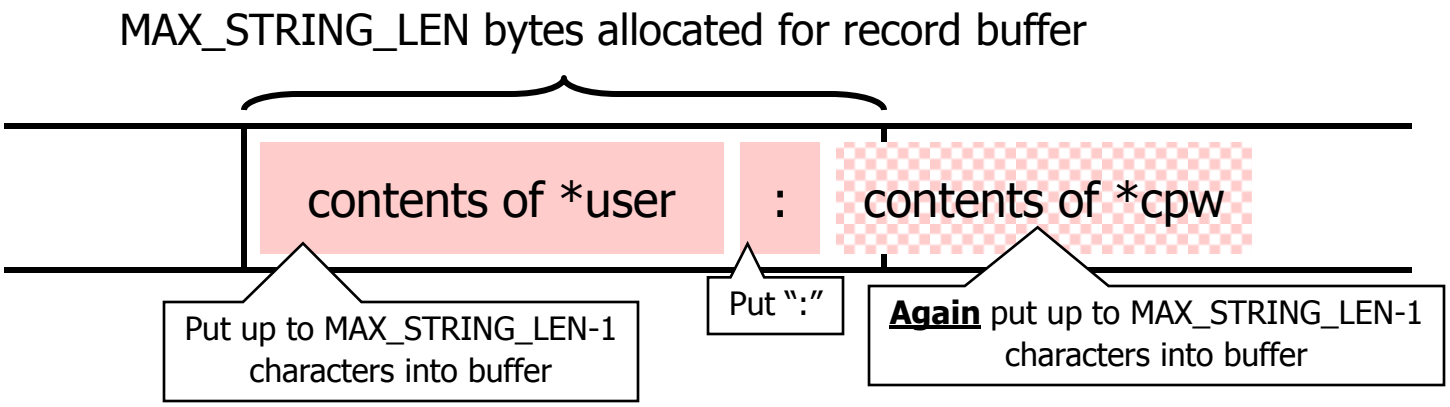
- Published "fix" (do you see the problem?)

```
… strncpy(record,user,MAX_STRING_LEN-1);
  strcat(record,":");
  strncat(record,cpw,MAX_STRING_LEN-1); …
```

Misuse of strncpy in htpasswd "Fix"

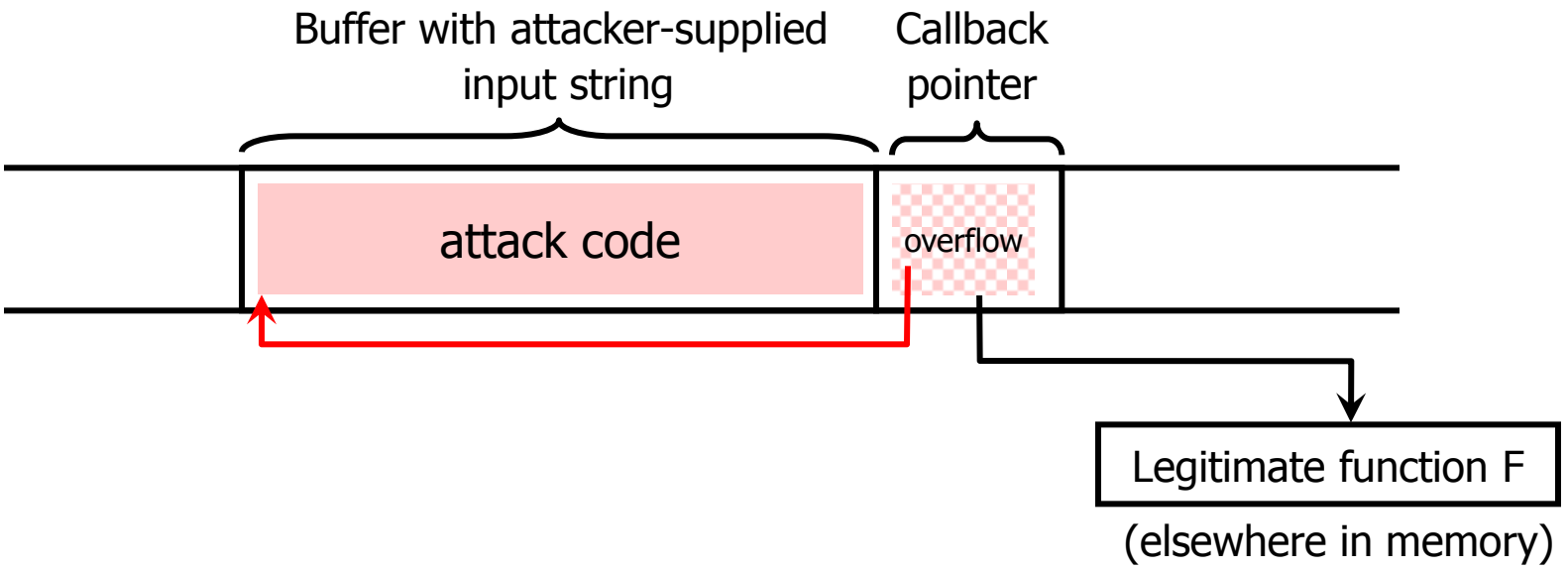
- Published "fix" for Apache htpasswd overflow:

```
… strncpy(record,user,MAX_STRING_LEN-1);
  strcat(record,":");
  strncat(record,cpw,MAX_STRING_LEN-1); …
```

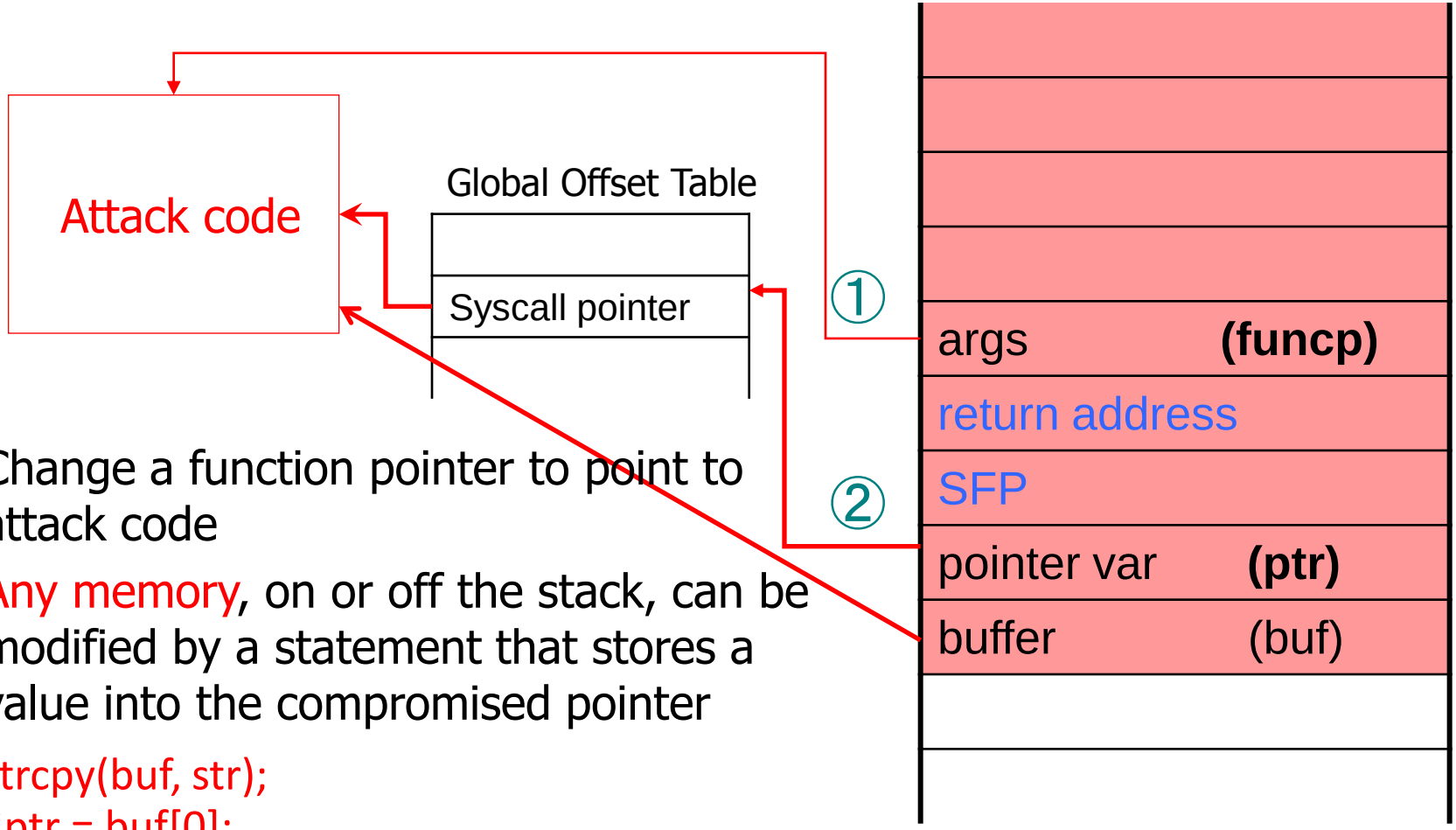


Function Pointer Overflow

- C uses function pointers for callbacks: if pointer to F is stored in memory location P, then another function G can call F as (*P)(...)



Attack #2: Pointer Variables



① Change a function pointer to point to attack code

② Any memory, on or off the stack, can be modified by a statement that stores a value into the compromised pointer

```
strcpy(buf, str);
*ptr = buf[0];
```

Off-By-One Overflow

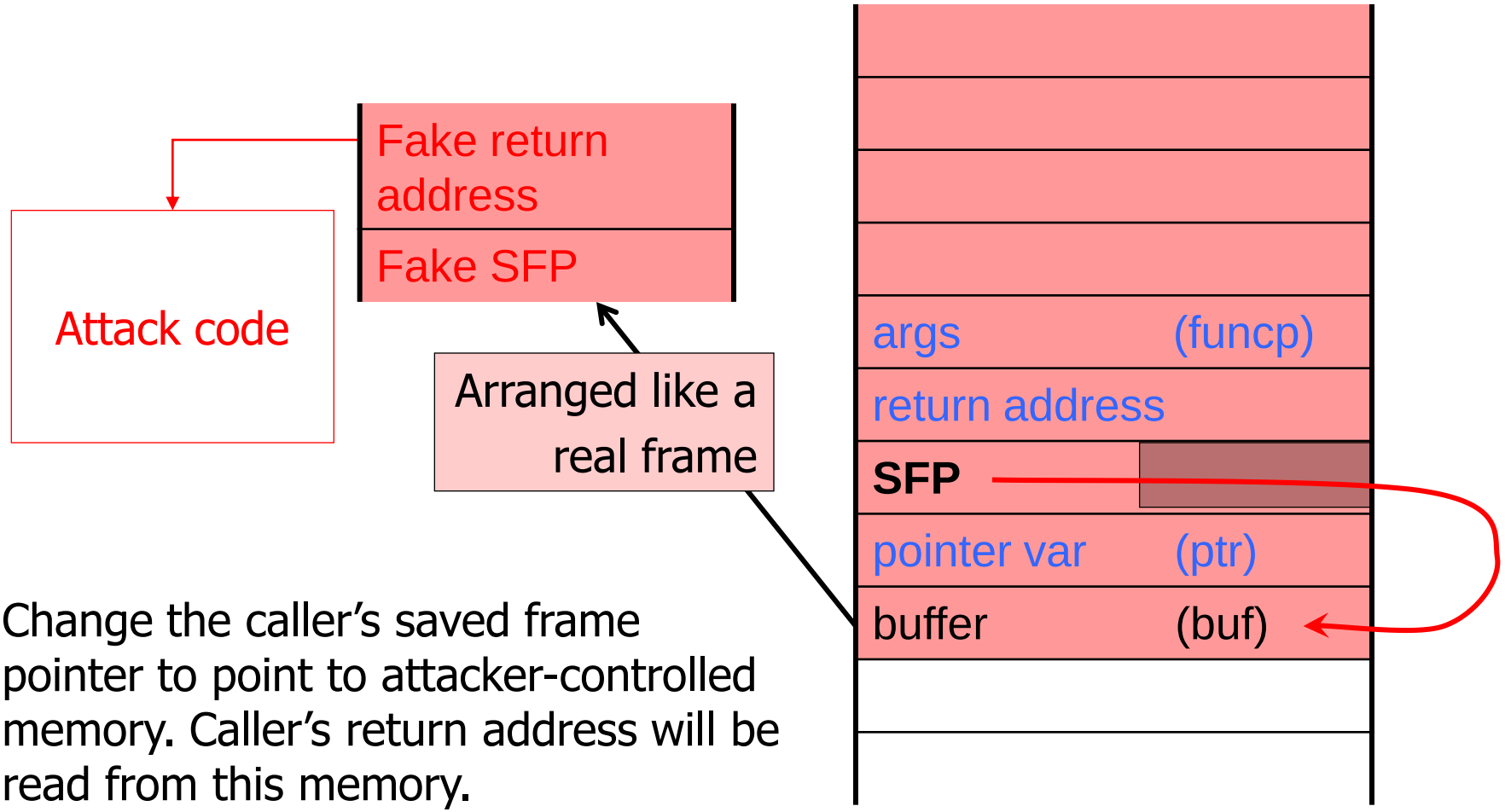
- Home-brewed range-checking string copy

```
void notSoSafeCopy(char *input) {
     char buffer[512]; int i;

     for (i=0; i<=512; i++)
         buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
     if (argc==2)
        notSoSafeCopy(argv[1]);
}
```

This will copy **513** characters into the buffer. Oops!

- 1-byte overflow: can't change RET, but can change saved pointer to previous stack frame
  - On little-endian architecture, make it point into buffer
  - Caller's RET will be read from buffer!

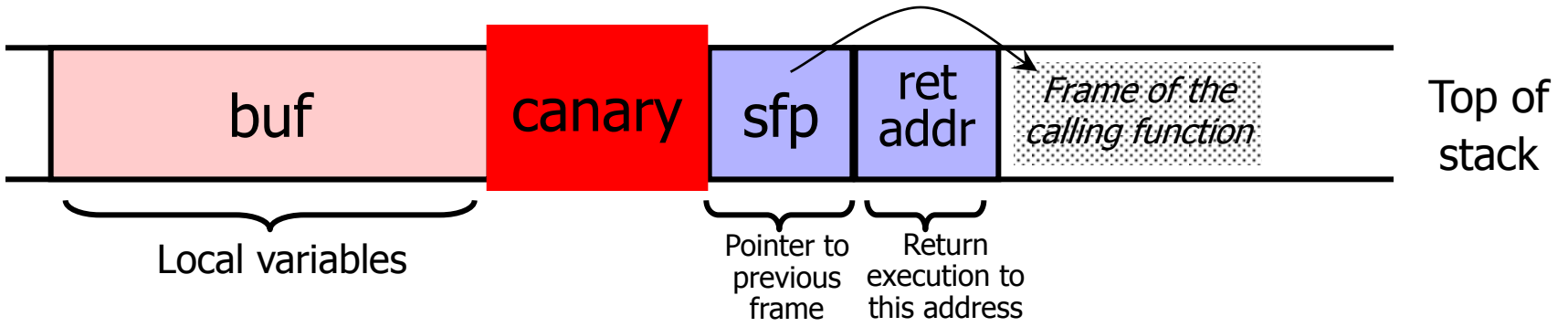
Attack #3: Frame Pointer



Change the caller's saved frame pointer to point to attacker-controlled memory. Caller's return address will be read from this memory.

Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



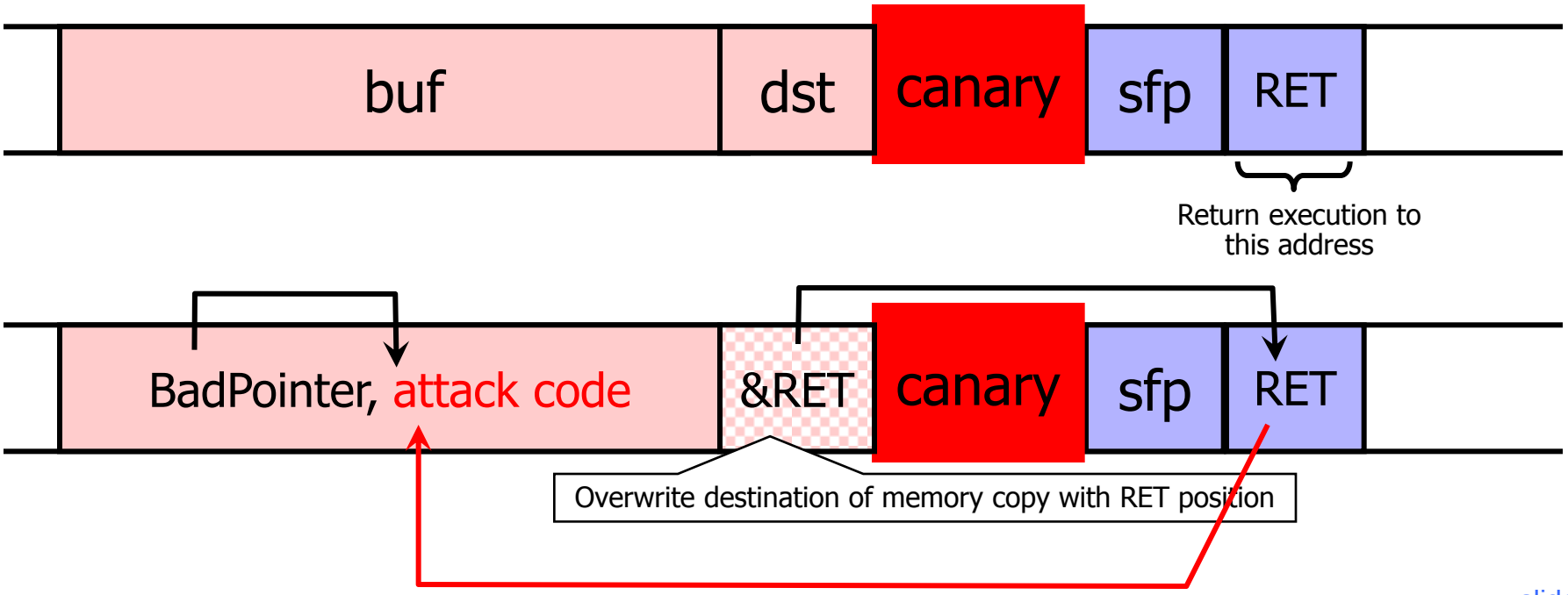
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- StackGuard can be defeated
  - A single memory copy where the attacker controls both the source and the destination is sufficient

Defeating StackGuard

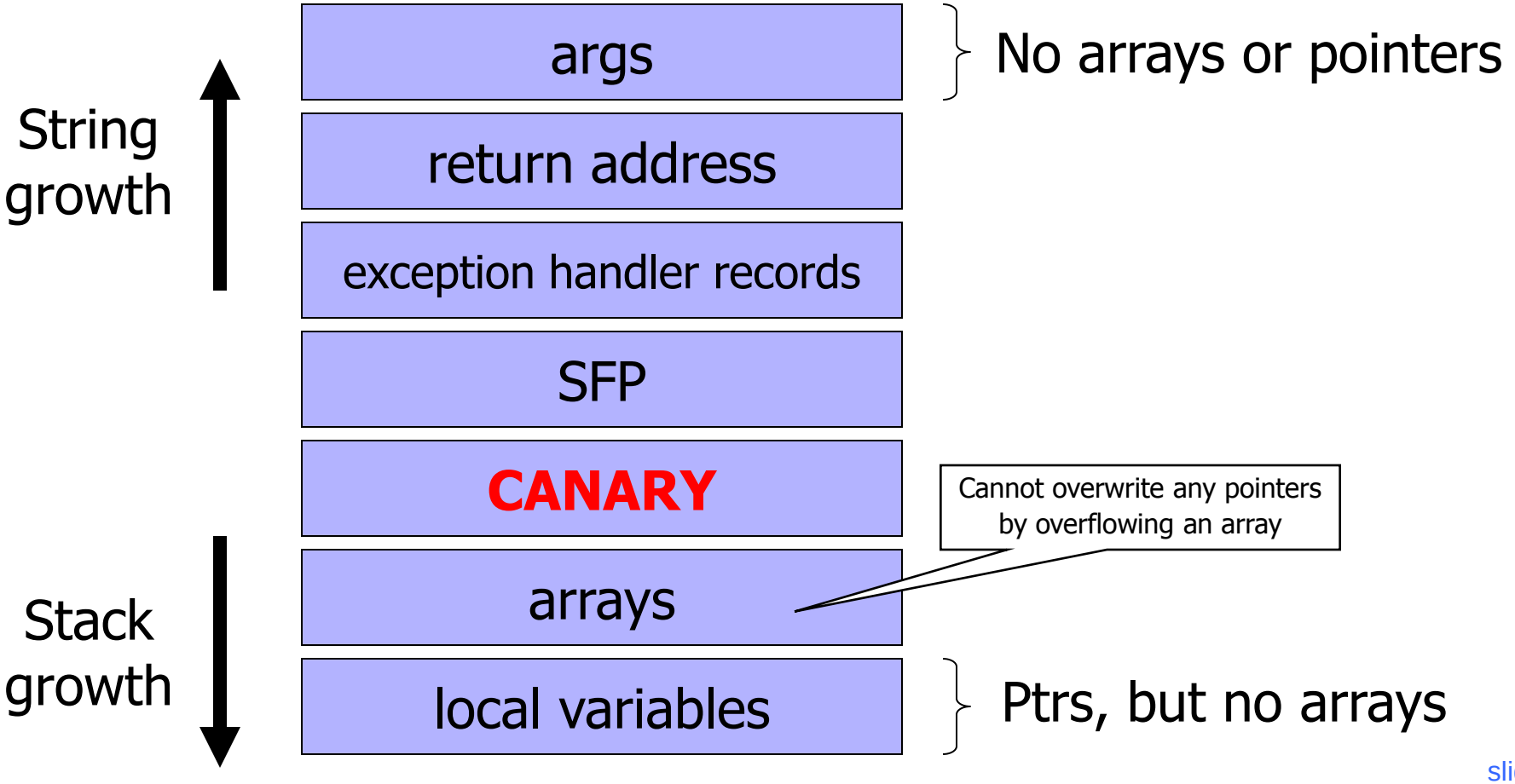
- Suppose program contains *dst=buf[0] where attacker controls both dst and buf
  - Example: dst is a local pointer variable



ProPolice / SSP

[IBM, used in gcc 3.4.1; also MS compilers]

- Rerrange stack layout (requires compiler mod)



What Can Still Be Overwritten?

- Other string buffers in the vulnerable function
- Any data stored on the stack
  - Exception handling records
  - Pointers to virtual method tables
    - C++: call to a member function passes as an argument "this" pointer to an object on the stack
    - Stack overflow can overwrite this object's vtable pointer and make it point into an attacker-controlled area
    - When a virtual function is called (how?), control is transferred to attack code (why?)
    - Do canaries help in this case?

      (Hint: when is the integrity of the canary checked?)

Litchfield's Attack

- Microsoft Windows 2003 server implements several defenses against stack overflow
  - Random canary (with /GS option in the .NET compiler)
  - When canary is damaged, exception handler is called
  - Address of exception handler stored on stack above RET
- Attack: smash the canary and overwrite the pointer to the exception handler with the address of the attack code
  - Attack code must be on heap and outside the module, or else Windows won't execute the fake "handler"
  - Similar exploit used by CodeRed worm

Safe Exception Handling

- Exception handler record must be on the stack of the current thread
- Must point outside the stack (why?)
- Must point to a valid handler
  - Microsoft's /SafeSEH linker option: header of the binary lists all valid handlers
- Exception handler records must form a linked list, terminating in FinalExceptionHandler
  - Windows Server 2008: SEH chain validation
  - Address of FinalExceptionHandler is randomized (why?)

When SafeSEH Is Incomplete

[Sotirov and Dowd]

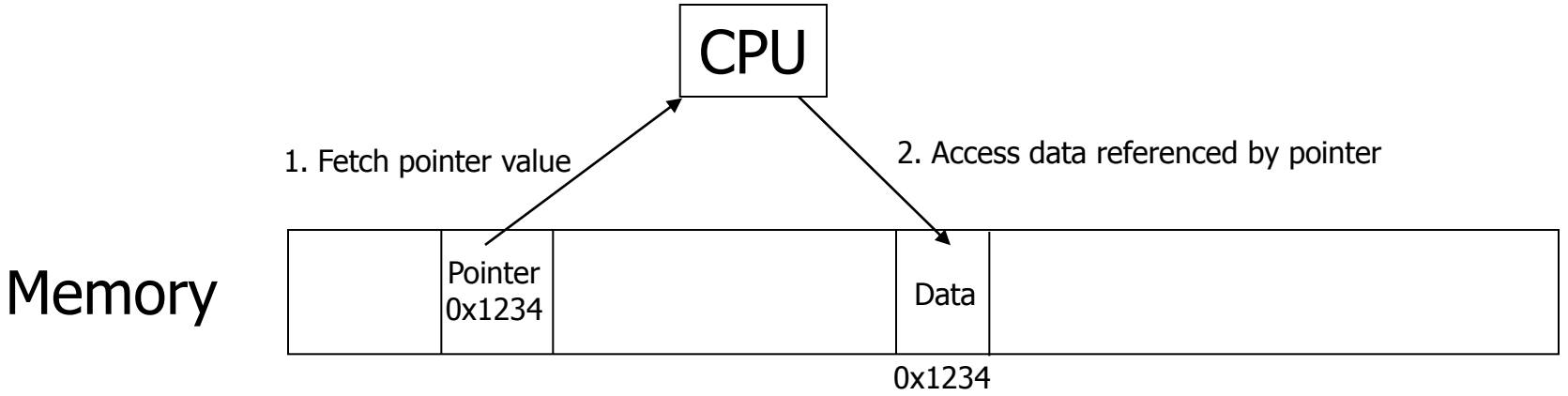
- If DEP is disabled, handler is allowed to be on any non-image page except stack
  - Put attack code on the heap, overwrite exception handler record on the stack to point to it
- If any module is linked without /SafeSEH, handler is allowed to be anywhere in this module
  - Overwrite exception handler record on the stack to point to a suitable place in the module

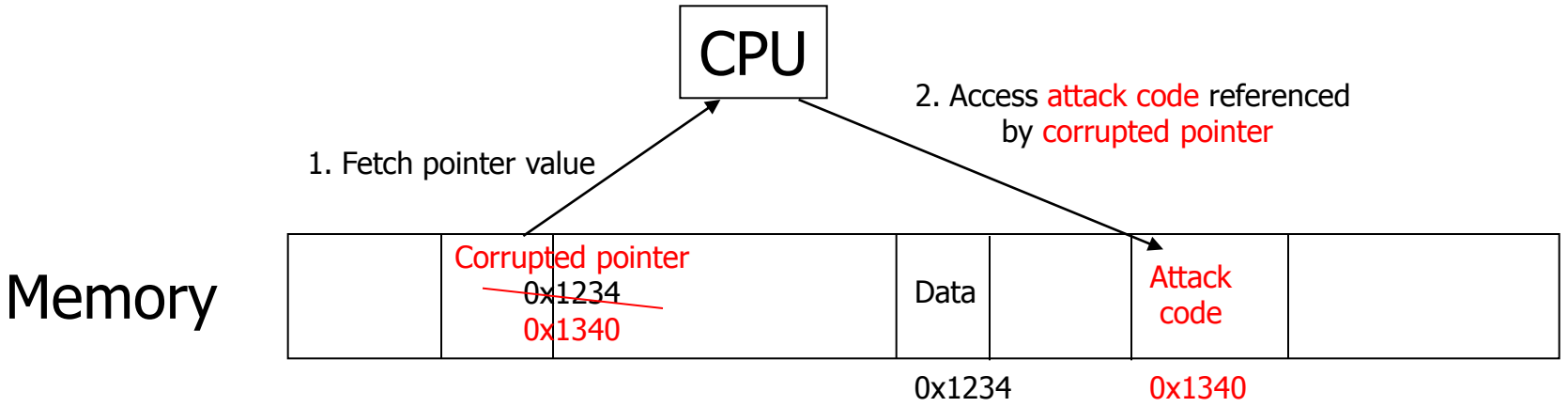
PointGuard

- Attack: overflow a function pointer so that it points to attack code
- Idea: encrypt all pointers while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    - Pointers cannot be overflown while in registers
- Attacker cannot predict the target program's key
  - Even if pointer is overwritten, after XORing with key it will dereference to a "random" memory address

Normal Pointer Dereference

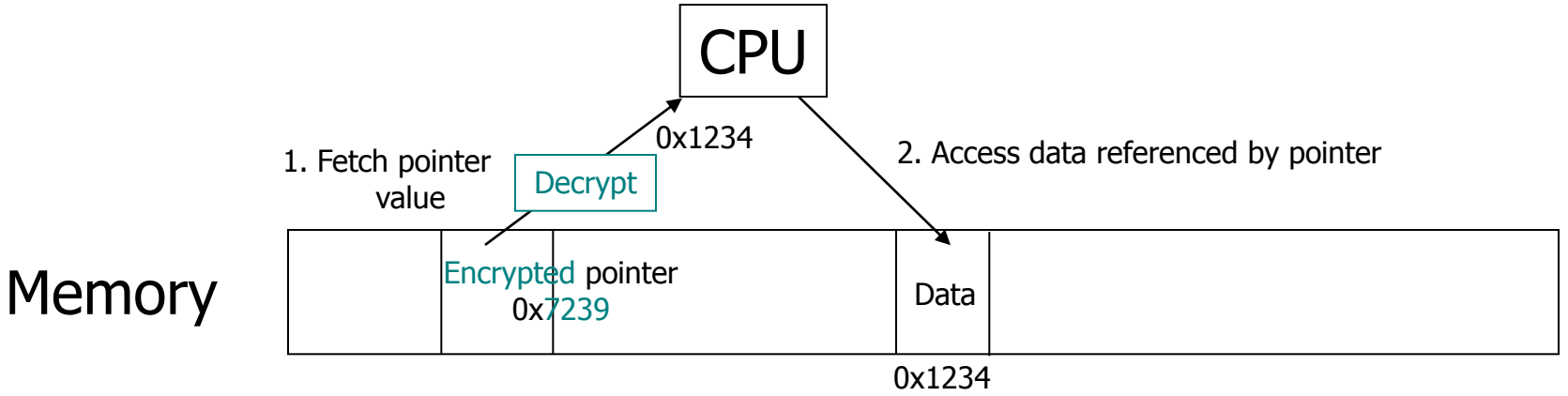
[Cowan]

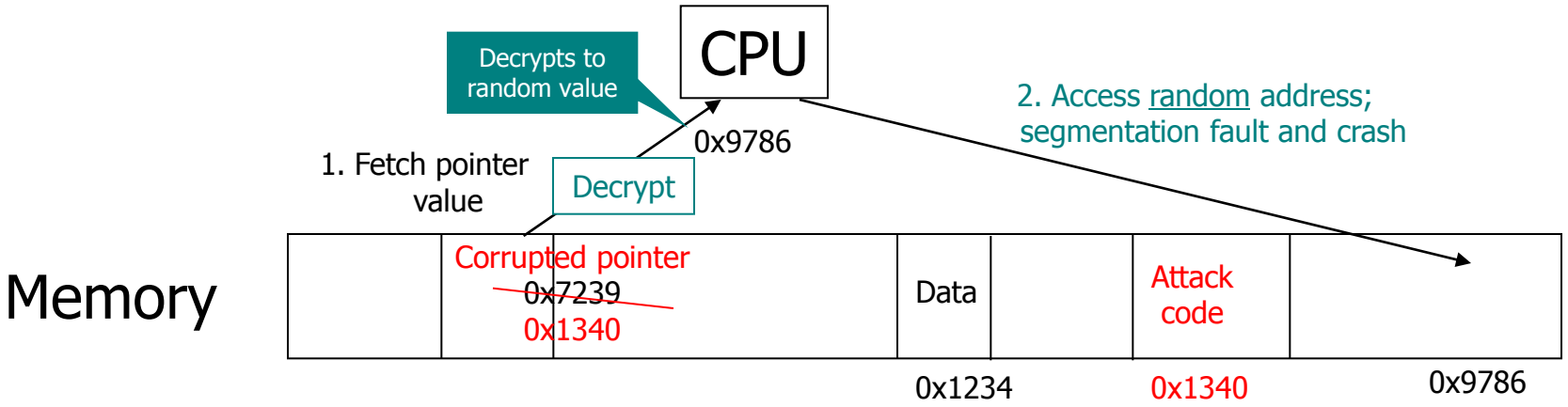




PointGuard Dereference

[Cowan]





PointGuard Issues

- Must be very fast
  - Pointer dereferences are very common
- Compiler issues
  - Must encrypt and decrypt only pointers
  - If compiler "spills" registers, unencrypted pointer values end up in memory and can be overwritten there
- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page
- PG'd code doesn't mix well with normal code
  - What if PG'd code needs to pass a pointer to OS kernel?

S. Chen et al.

Non-Control-Data Attacks Are Realistic Threats

(USENIX Security 2005)

Non-Control Targets

[Chen et al.]

- Configuration parameters
  - Example: directory names that confine remotely invoked programs to a portion of the file system
- Pointers to names of system programs
  - Example: replace the name of a harmless script with an interactive shell
  - This is not the same as return-to-libc (why?)
- Branch conditions in input validation code

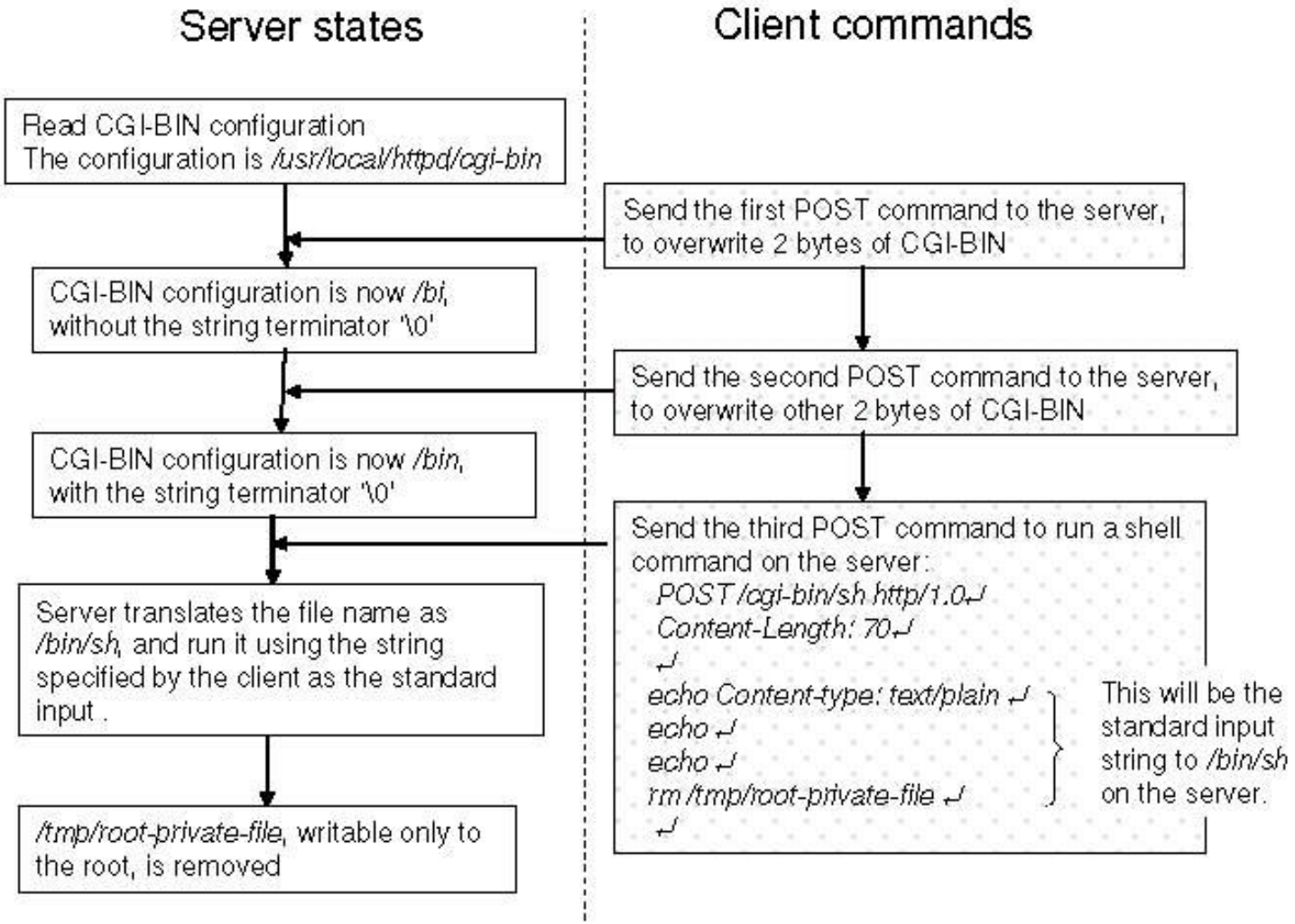
Example: Web Server Security

- CGI scripts are executables on Web server that can be executed by remote user via a special URL
  - http://www.server.com/cgi-bin/SomeProgram
- Don't want remote users executing arbitrary programs with the Web server's privileges
  - Need to restrict which programs can be executed
- CGI-BIN is the directory name which is always prepended to the name of the CGI script
  - If CGI-BIN is "/usr/local/httpd/cgi-bin", the above URL will execute /usr/local/httpd/cgi-bin/SomeProgram

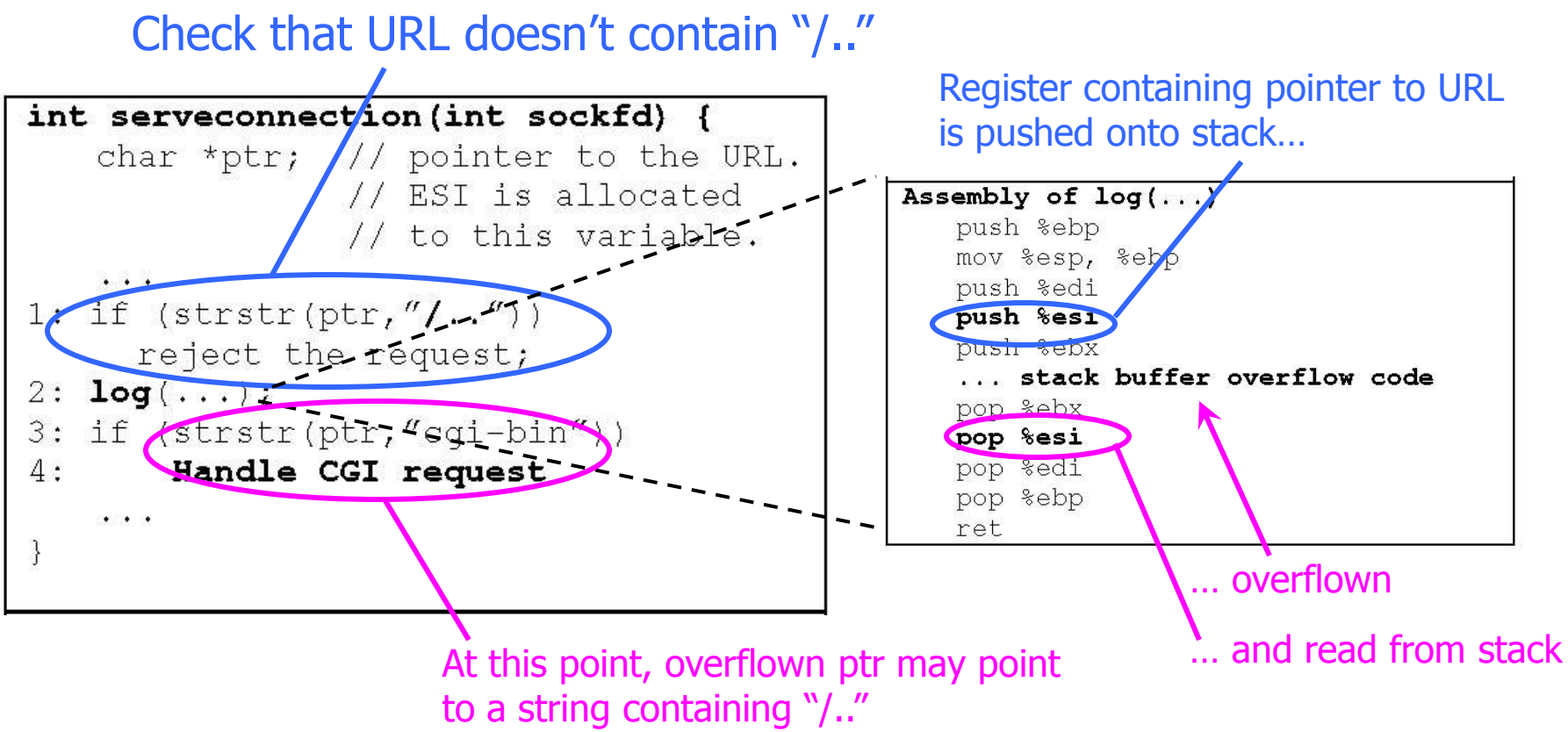
Exploiting Null HTTP Heap Overflow

- Null HTTPD had a heap overflow vulnerability
  - When the corrupted buffer is freed, an overflown value is copied to a location whose address is read from an overflown memory area
  - This enables attacker to copy an arbitrary value into a memory location of his choice
- Standard exploit: copy address of attack code into the table containing addresses of library functions
  - Transfers control to attacker's code next time the library function is called
- Alternative: overwrite the value of CGI-BIN

Null HTTP CGI-BIN Exploit



Another Web Server: GHTTPD



ptr changes after it was checked

but before it was used! (time-of-check-to-time-of-use attack)

SSH Authentication Code

```
void do_authentication(char *user, ...) {
1:   int authenticated = 0;
     ...
2:   while (!authenticated) {
       /* Get a packet from the client */
3:     type = packet_read();
       /* calls detect_attack() internally
4:     switch (type) {
       ...
5:     case SSH_CMSG_AUTH_PASSWORD:
6:      if (auth_password(user, password))
7:           authenticated =1;
       case ...
       }
8:     if (authenticated) break;
     }
     /* Perform session preparation. */
9:   do_authenticated(pw);
}
```

write 1 here

Loop until one of the authentication methods succeeds

detect_attack() prevents checksum attack on SSH1…

…and also contains an overflow bug which permits the attacker to put any value into any memory location

Break out of authentication loop without authenticating properly

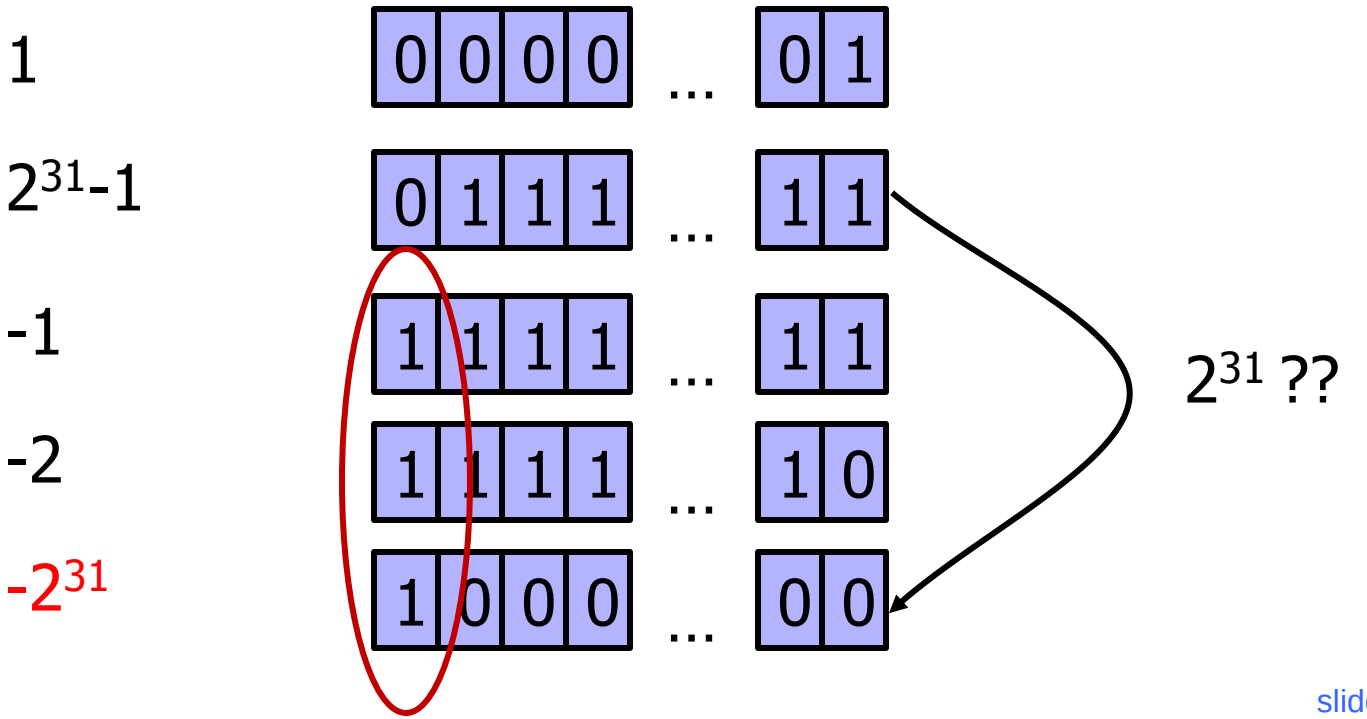
Reducing Lifetime of Critical Data

```
(B2)Modified SSHD do_authentication()
{ int authenticated = 0;
  while (!authenticated) {
L1:type = packet_read(); //vulnerable
   authenticated = 0;
   switch (type) {
    case SSH_CMSG_AUTH_PASSWORD:
     if (auth_password(user, passwd))
          authenticated = 1;
    case ...
   }
   if (authenticated) break;
  }
  do_authenticated(pw);
}
```

Reset flag here, right before doing the checks

Two's Complement

- Binary representation of negative integers
- Represent X (where X<0) as 2^N-|X|
  - N is word size (e.g., 32 bits on x86 architecture)



Integer Overflow

```
static int getpeername1(p, uap, compat) {
// In FreeBSD kernel, retrieves address of peer to which a socket is connected
    …
    struct sockaddr *sa;
     …
    len = MIN(len, sa->sa_len);
    … copyout(sa, (caddr_t)uap->asa, (u_int)len);
    …
}
```

Checks that "len" is not too big

Negative "len" will always pass this check…

… interpreted as a huge unsigned integer here

Copies "len" bytes from kernel memory to user space

… will copy up to 4G of kernel memory

M. Dowd

Application-Specific Attacks: Leveraging the ActionScript Virtual Machine

(IBM X-Force report 2008)

ActionScript Exploit

[Dowd]

- ActionScript 3 is a scripting language for Flash
  - Basically, JavaScript for Flash animations
  - For performance, Flash 9 and higher compiles scripts into bytecode for ActionScript Virtual Machine (AVM2)
- Flash plugins are installed on millions of browsers, thus a perfect target for attack
  - Different Flash binaries are used for Internet Explorer and Firefox, but this turns out not to matter

Processing SWF Scene Records (1)

Code that allocates memory for scene records:

Supplied as part of SWF file from potentially malicious website

```
call   SWF_GetEncodedInteger ; Scene Count
mov  edi, [ebp+arg_0]
mov  [esi+4], eax      ← How much memory is needed to store scenes
mov  ecx, [ebx+8]      ← Total size of the buffer
sub   ecx, [ebx+4]     ← Offset into the buffer
cmp    eax, ecx        ← Is there enough memory in the buffer?
jg      loc_30087BB4   ← (signed comparison)
...
push  eax              ← Tell mem_Calloc how many bytes to allocate
call    mem_Calloc     ← Interprets its argument as unsigned integer
```

What if scene count is negative?

mem_Calloc fails (why?) and returns NULL

Processing SWF Scene Records (2)

- Scene records are copied as follows:
  - Start with pointer P returned by allocator
  - Loop through and copy scenes until count ≤ 0
  - Copy frame count into P + offset, where offset is determined by scene count
    - Frame count also comes from the SWF file
    - It is a "short" (16-bit) value, but written as a 32-bit DWORD
- Attacker gains the ability to write one value into any location in memory (why?)
  - … subject to some restrictions (see paper)
  - But this is not enough to hijack control directly (why?)

ActionScript Virtual Machine (AVM2)

- Register-based VM
  - Bytecode instructions write and read from "registers"
- "Registers", operand stack, scope stack allocated on the same runtime stack as used by Flash itself
  - "Registers" are mapped to locations on the stack and accessed by index (converted into memory offset)
  - This is potentially dangerous (why?)
- Malicious Flash script could hijack browser's host
  - Malicious bytecode can write into any location on the stack by supplying a fake register index
  - This would be enough to take control (how?)

AVM2 Verifier

- ActionScript code is verified before execution
- All bytecodes must be valid
  - Throw an exception if encountering an invalid bytecode
- All register accesses correspond to valid locations on the stack to which registers are mapped
- For every instruction, calculate the number of operands, ensure that operands of correct type will be on the stack when it is executed
- All values are stored with correct type information
  - Encoded in bottom 3 bits

Relevant Verifier Code

```
…
if(AS3_argmask[opCode] == 0xFF) {
   … throw exception …
}
…
opcode_getArgs(…)
…

void opcode_getArgs(…) {
   DWORD mask=AS3_argmask[opCode];
   …
   if(mask <=0) { … return … }
   … *arg_dword1 = SWF_GetEncodedInteger(&ptr);
   if(mask>1) *arg_dword2 = SWF_GetEncodedInteger(&ptr);
}
```

Invalid bytecode

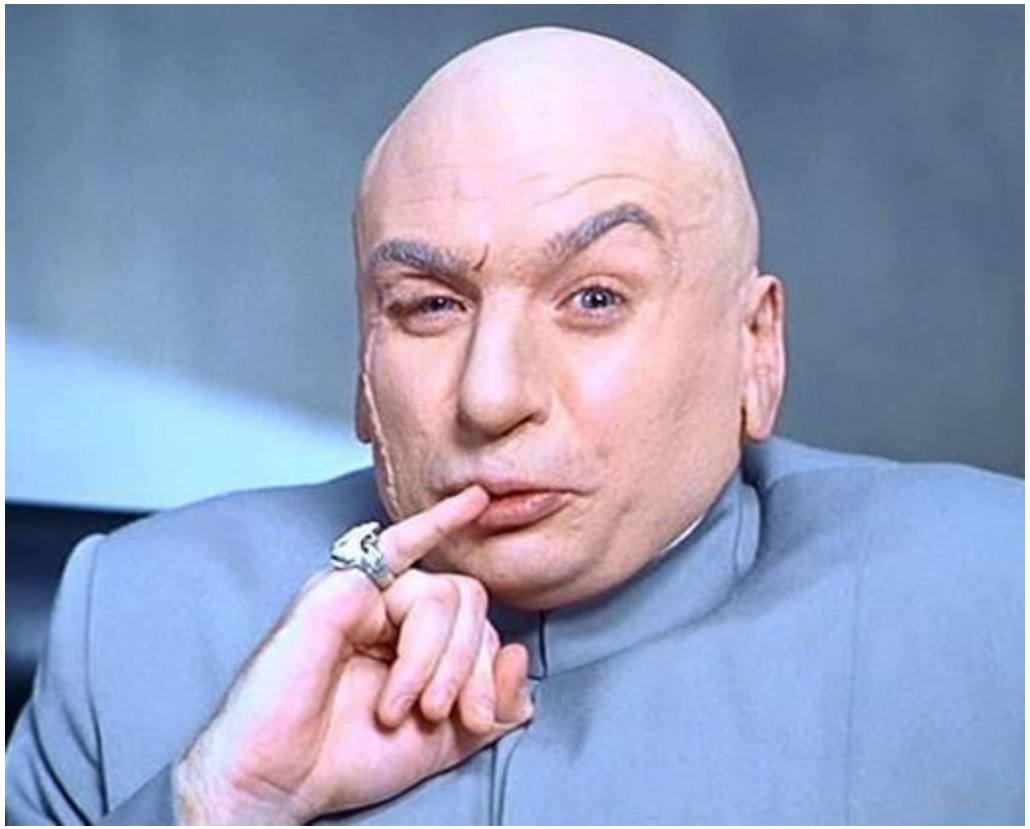
Number of operands for each opcode is defined in AS3_argmask array

Determine operands

Executing Invalid Opcodes

- If interpreter encounters an invalid opcode, it silently skips it and continues executing
  - Doesn't really matter because this can't happen
    - Famous last words…
  - AS3 code is executed only after it has been verified, and verifier throws an exception on invalid bytecode
- But if we could somehow trick the verifier…
  - Bytes after the opcode are treated as data (operands) by the verifier, but as executable code by interpreter
  - This is an example of a TOCTTOU (time-of-check-to-time-of-use) vulnerability

Breaking AVM2 Verifier



Breaking AVM2 Verifier

- Pick an invalid opcode
- Use the ability to write into arbitrary memory to change the AS3_argmask of that opcode from 0xFF to something else
- AVM2 verifier will treat it as normal opcode and skip subsequent bytes as operands
  - How many? This is also determined by AS3_argmask!
- AVM2 interpreter, however, will skip the invalid opcode and execute those bytes
- You can now execute unverified ActionScript code

Further Complications

- Can execute only a few unverified bytecodes at a time (why?)
  - Use multiple "marker" opcodes with overwritten masks
- Cannot directly overwrite saved EIP on the evaluation stack with the address of shellcode because 3 bits are clobbered by type information
  - Stack contains a pointer to current bytecode (codePtr)
  - Move it from one "register" to another, overwrite EIP
  - Bytecode stream pointed to by codePtr contains a jump to the actual shellcode
- Read the paper for more details