

Buffer Overflow and Other Memory Corruption Attacks

Vitaly Shmatikov

Reading Assignment

- ◆ You MUST read **Smashing the Stack for Fun and Profit** to understand how to start on the project
- ◆ Read **Once Upon a free()**
 - Also on malloc() exploitation: **Vudo - An Object Superstitiously Believed to Embody Magical Powers**
- ◆ Read **Exploiting Format String Vulnerabilities**
- ◆ Optional reading
 - **Blended Attacks** by Chien and Szor to better understand how overflows are used by malware
 - **The Tao of Windows Buffer Overflow** as taught by DilDog from the Cult of the Dead Cow

Morris Worm



- ◆ Released in 1988 by Robert Morris
 - Graduate student at Cornell, son of NSA chief scientist
 - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
 - Now a computer science professor at MIT
- ◆ Morris claimed it was intended to harmlessly measure the Internet, but it created new copies as fast as it could and overloaded infected hosts
- ◆ \$10-100M worth of damage

Morris Worm and Buffer Overflow

- ◆ We will look at the Morris worm in more detail when talking about worms and viruses
- ◆ One of the worm's propagation techniques was a buffer overflow attack against a vulnerable version of fingerd on VAX systems
 - By sending a special string to finger daemon, worm caused it to execute code creating a new worm copy
 - Unable to determine remote OS version, worm also attacked fingerd on Suns running BSD, causing them to crash (instead of spawning a new copy)

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
 - Windows LNK shortcut display
 - Windows task scheduler
- ◆ Flame (2010-12): same print spooler and LNK overflows as Stuxnet
 - Targeted cyberespionage virus

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
 - Many attacks do not involve executing “data”
- ◆ Attack can exploit any memory operation
 - Pointer assignment, format strings, memory allocation and de-allocation, function pointers, calls to library routines via offset tables ...

Stack Buffers

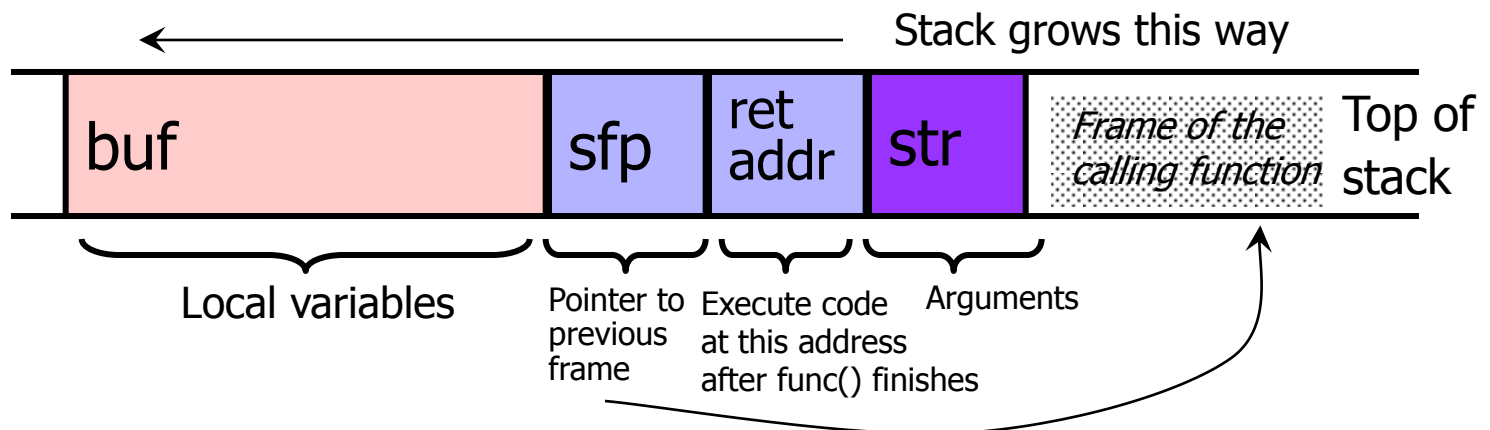
- ◆ Suppose Web server contains this function

```
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



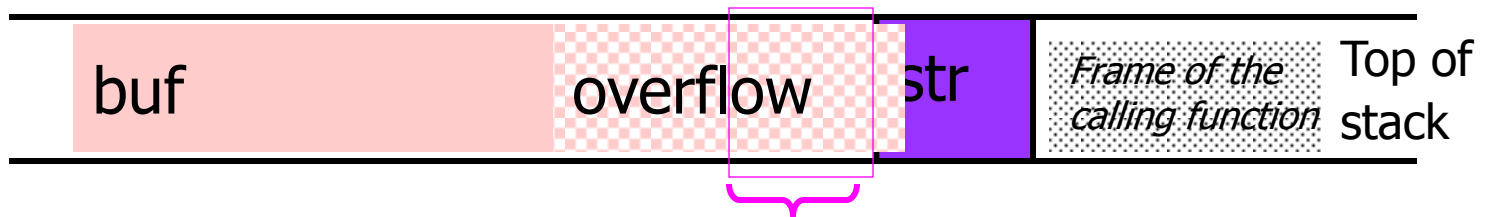
What If Buffer Is Overstuffed?

- ◆ 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

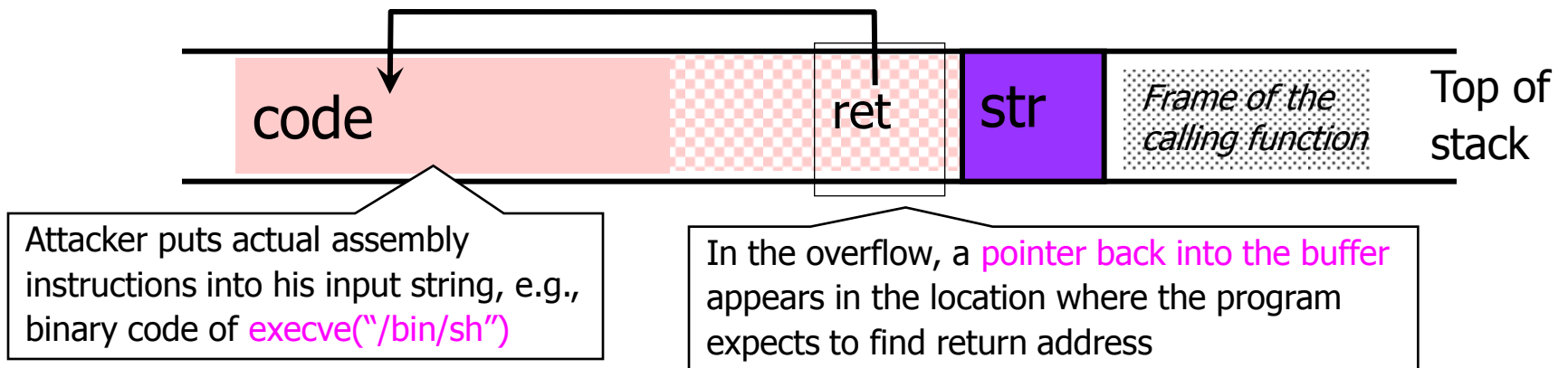
- ◆ If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations



This will be interpreted as return address!

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`

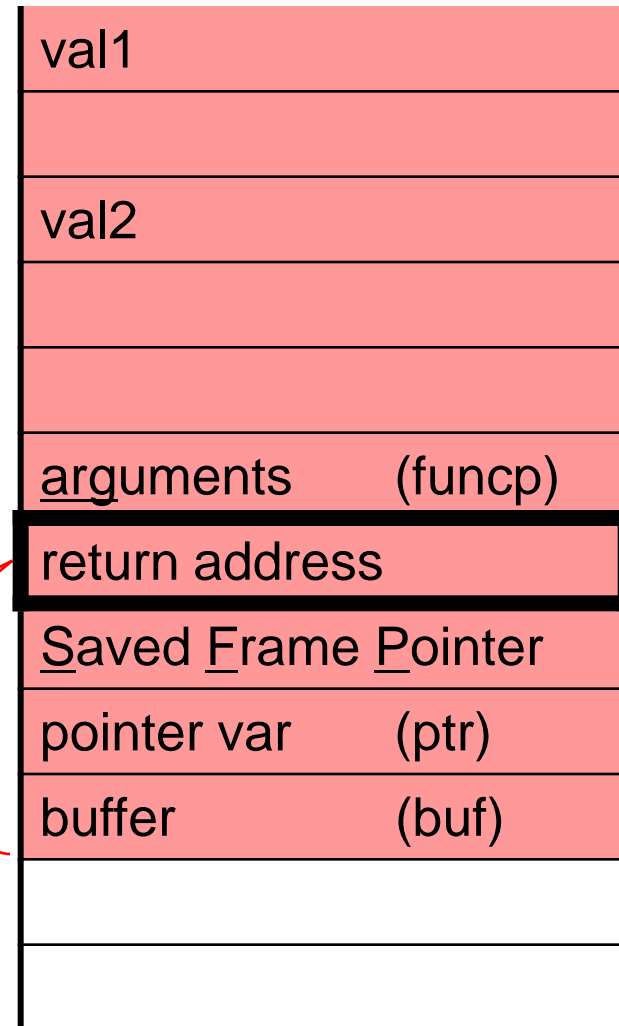
Stack Corruption: General View

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

Attacker-controlled memory

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

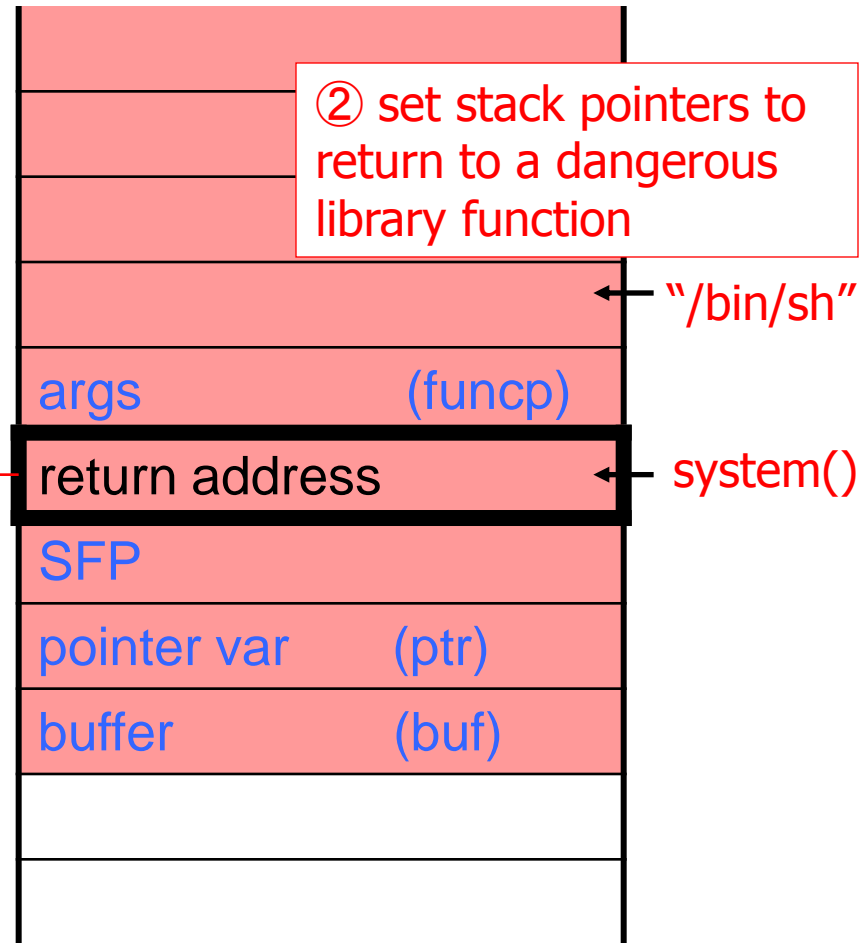
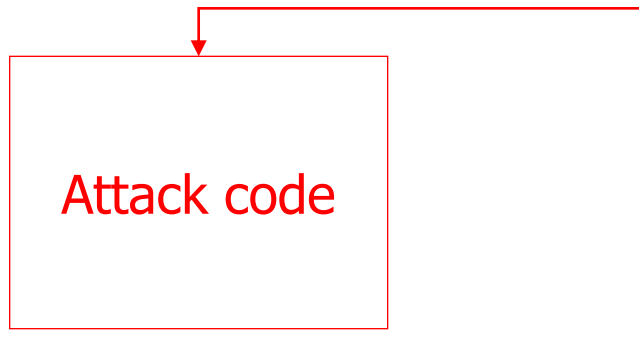
Most popular target



String grows

Stack grows

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.

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

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`
- ◆ Standard C library functions are all 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?):

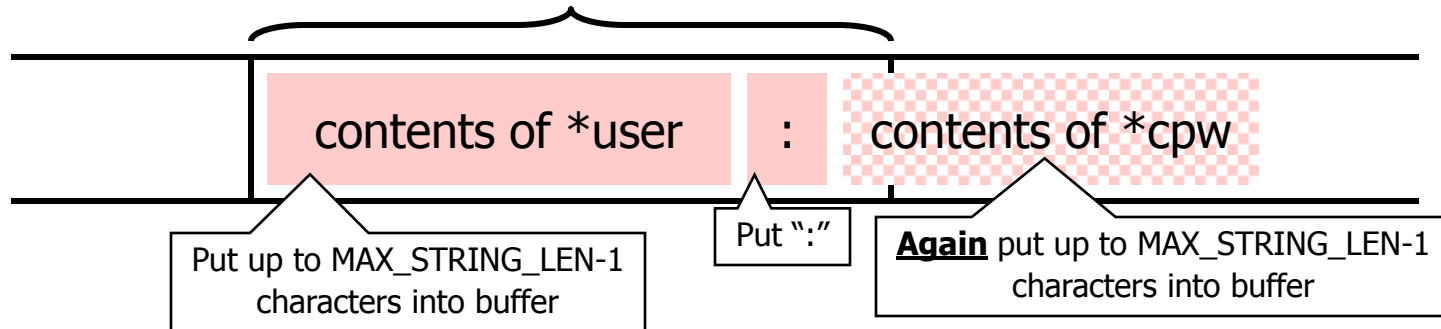
```
... strncpy(record, user, MAX_STRING_LEN-1) ;  
   strcat(record, ":" ) ;  
   strncpy(record, cpw, MAX_STRING_LEN-1) ; ...
```

Misuse of strncpy in httpasswd "Fix"

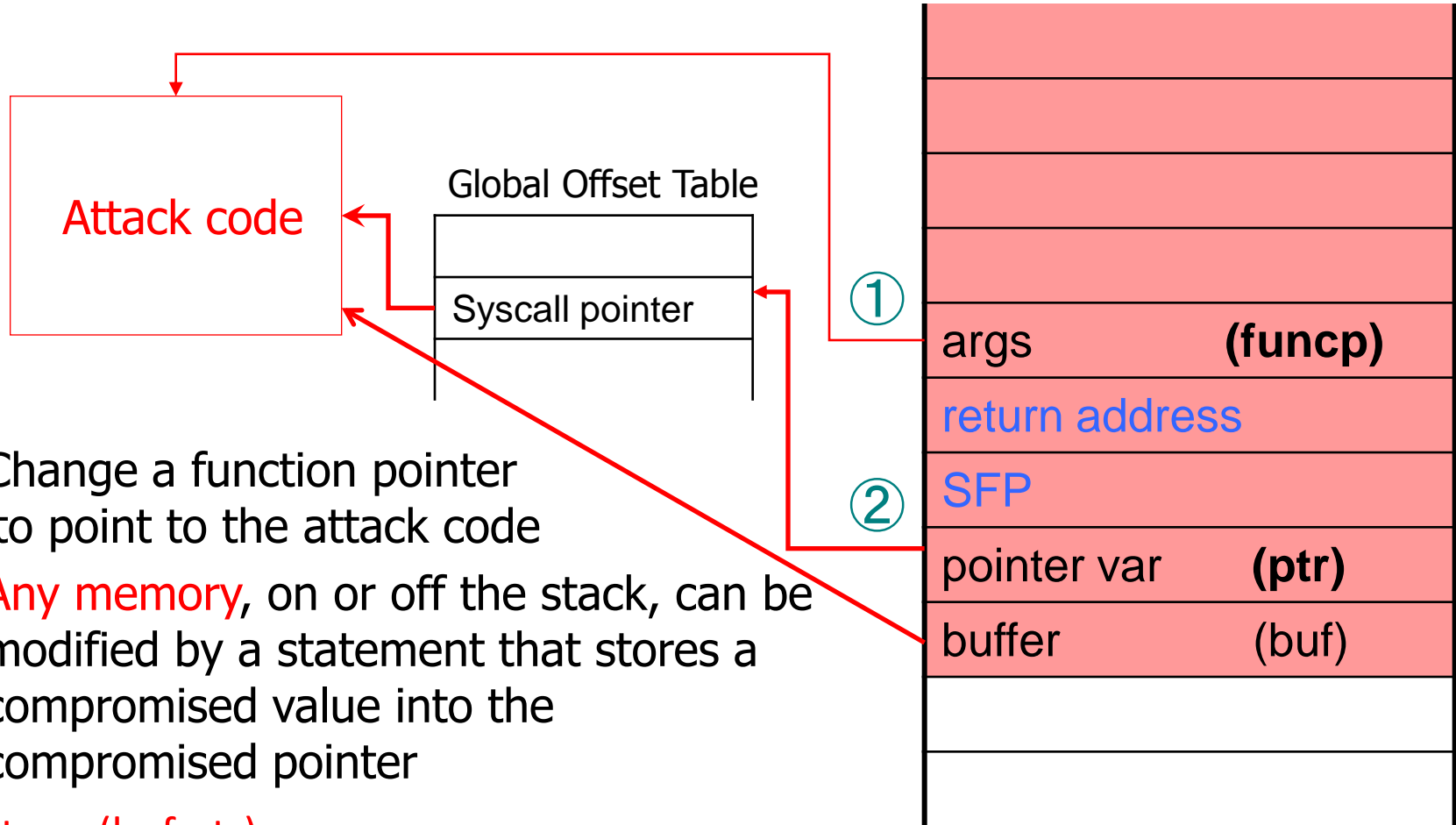
◆ Published "fix" for Apache httpasswd overflow:

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

MAX_STRING_LEN bytes allocated for record buffer



Attack #2: Pointer Variables



- ① Change a function pointer to point to the attack code
- ① **Any memory**, on or off the stack, can be modified by a statement that stores a compromised value into the compromised pointer

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

Off-By-One Overflow

◆ Home-brewed range-checking string copy

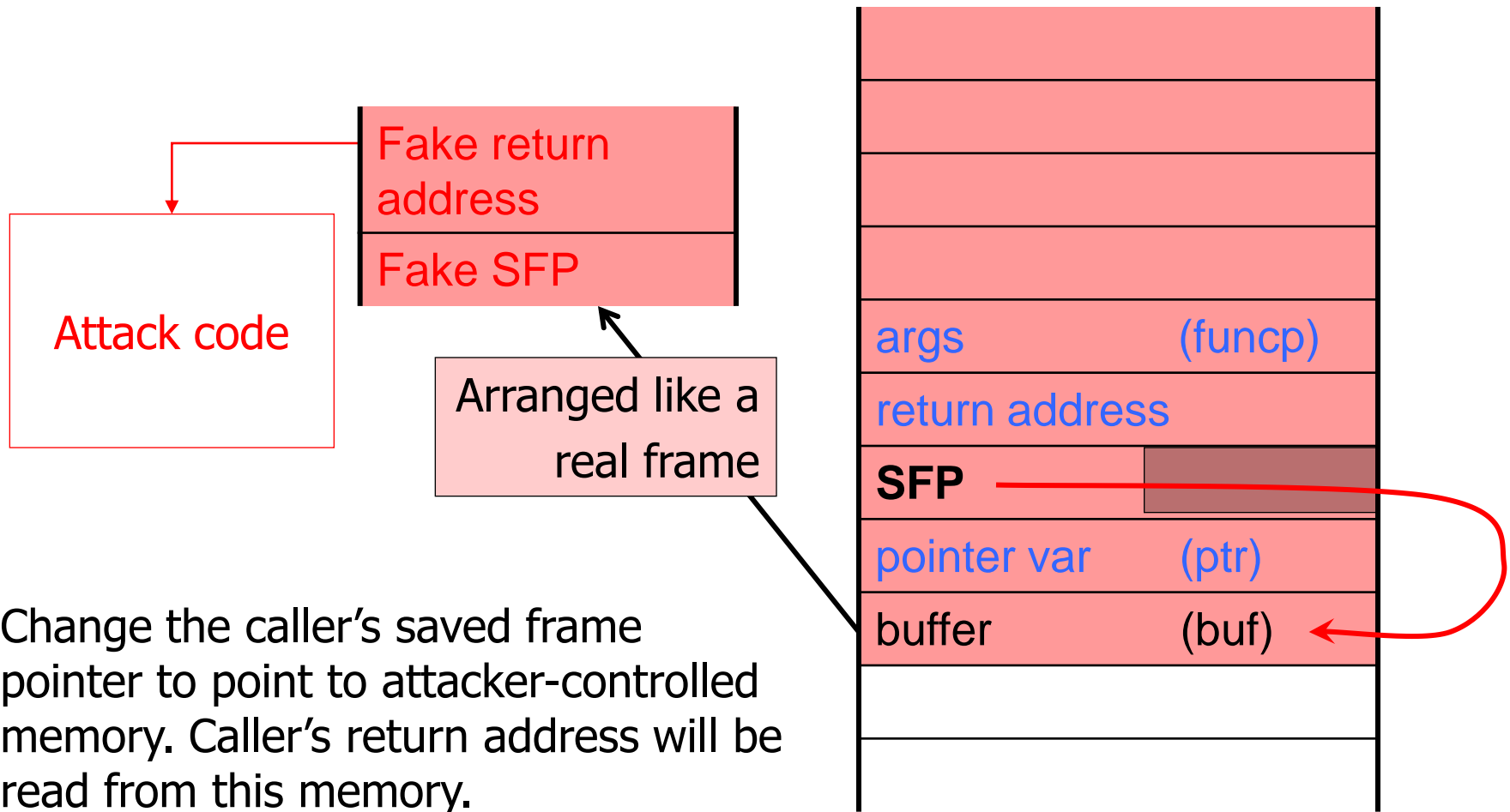
```
void notSoSafeCopy(char *input) {
    char buffer[512]; int i;
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
    if (argc==2)
        notSoSafeCopy(argv[1]);
}
```

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

Attack #3: Frame Pointer



Buffer Overflow: Causes and Cures

- ◆ Typical memory exploit involves **code injection**
 - Put malicious code at a predictable location in memory, usually masquerading as data
 - Trick vulnerable program into passing control to it
 - Overwrite saved EIP, function callback pointer, etc.
- ◆ Idea: **prevent execution of untrusted code**
 - Make stack and other data areas non-executable
 - Note: messes up useful functionality (e.g., Flash, JavaScript)
 - Digitally sign all code
 - Ensure that all control transfers are into a trusted, approved code image

W⊕X / DEP

- ◆ Mark all writeable memory locations as non-executable
 - Example: Microsoft's Data Execution Prevention (DEP)
 - This blocks (almost) all code injection exploits
- ◆ Hardware support
 - AMD "NX" bit, Intel "XD" bit (in post-2004 CPUs)
 - Makes memory page non-executable
- ◆ Widely deployed
 - Windows (since XP SP2),
Linux (via PaX patches),
OS X (since 10.5)



What Does $W\oplus X$ Not Prevent?

- ◆ Can still corrupt stack ...
 - ... or function pointers or critical data on the heap
- ◆ As long as “saved EIP” points into existing code, $W\oplus X$ protection will not block control transfer
- ◆ This is the basis of **return-to-libc** exploits
 - Overwrite saved EIP with address of any library routine, arrange stack to look like arguments
- ◆ Does not look like a huge threat
 - Attacker cannot execute arbitrary code, especially if `system()` is not available

return-to-libc on Steroids

- ◆ Overwritten saved EIP need not point to the beginning of a library routine
- ◆ Any existing instruction in the code image is fine
 - Will execute the sequence starting from this instruction
- ◆ What if instruction sequence contains RET?
 - Execution will be transferred... to where?
 - Read the word pointed to by stack pointer (ESP)
 - Guess what? Its value is under attacker's control! (why?)
 - Use it as the new value for EIP
 - Now control is transferred to an address of attacker's choice!
 - Increment ESP to point to the next word on the stack

Chaining RETs for Fun and Profit

[Shacham et al.]

- ◆ Can chain together sequences ending in RET
 - Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)
- ◆ What is this good for?
- ◆ Answer [Shacham et al.]: **everything**
 - Turing-complete language
 - Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
 - Attack can perform arbitrary computation using no injected code at all – **return-oriented programming**

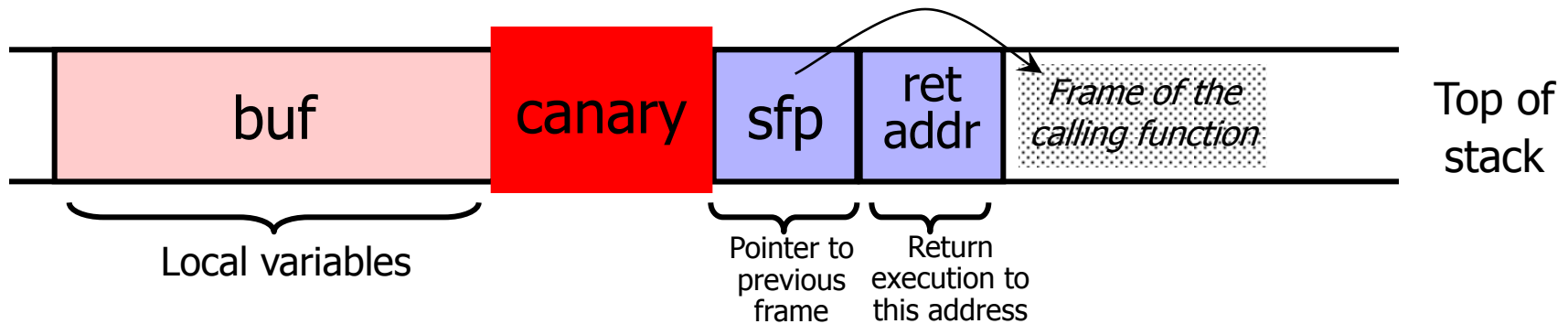


Other Issues with W \oplus X / DEP

- ◆ Some applications require executable stack
 - Example: Flash ActionScript, Lisp, other interpreters
- ◆ Some applications are not linked with /NXcompat
 - DEP disabled (e.g., some Web browsers)
- ◆ JVM makes all its memory RWX – readable, writable, executable (**why?**)
 - Spray attack code over memory containing Java objects (how?), pass control to them
- ◆ “Return” into a memory mapping routine, make page containing attack code writeable

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



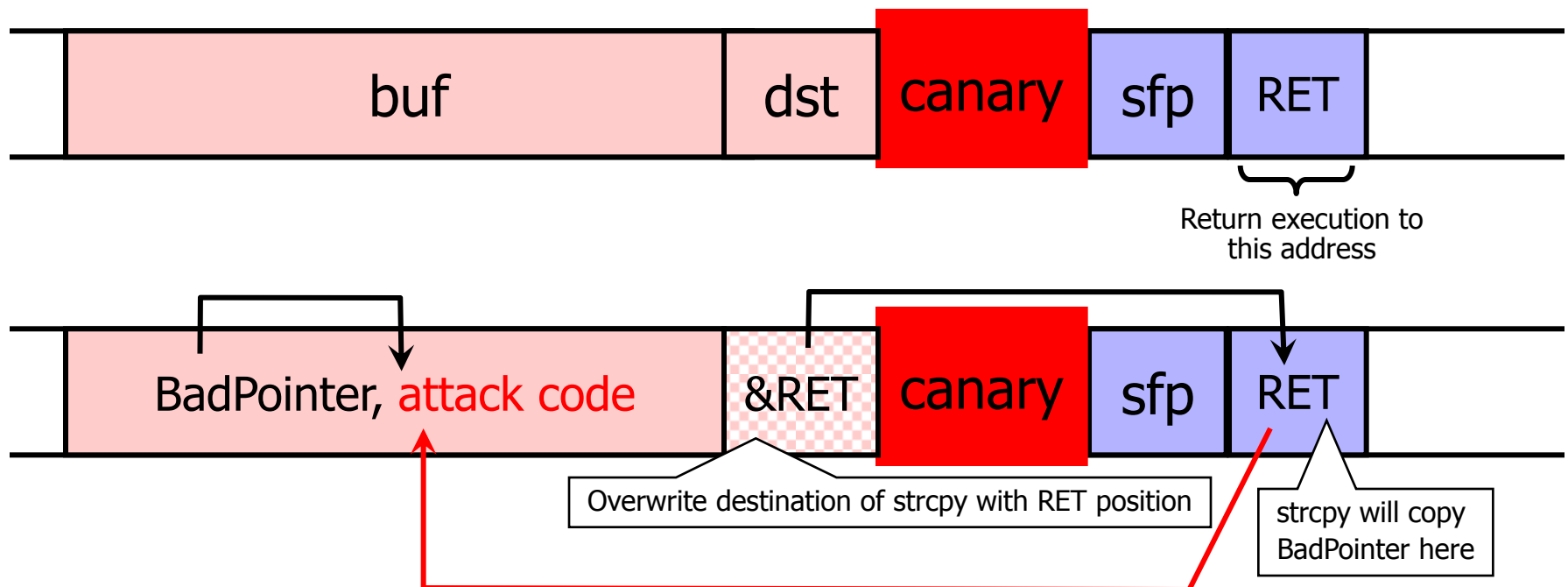
- ◆ Choose random canary string on program start
 - Attacker can't guess what the value of canary will be
- ◆ Terminator canary: “\0”, newline, linefeed, EOF
 - String functions like strcpy won't copy beyond “\0”

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 write where the attacker controls both the value and the destination is sufficient

Defeating StackGuard

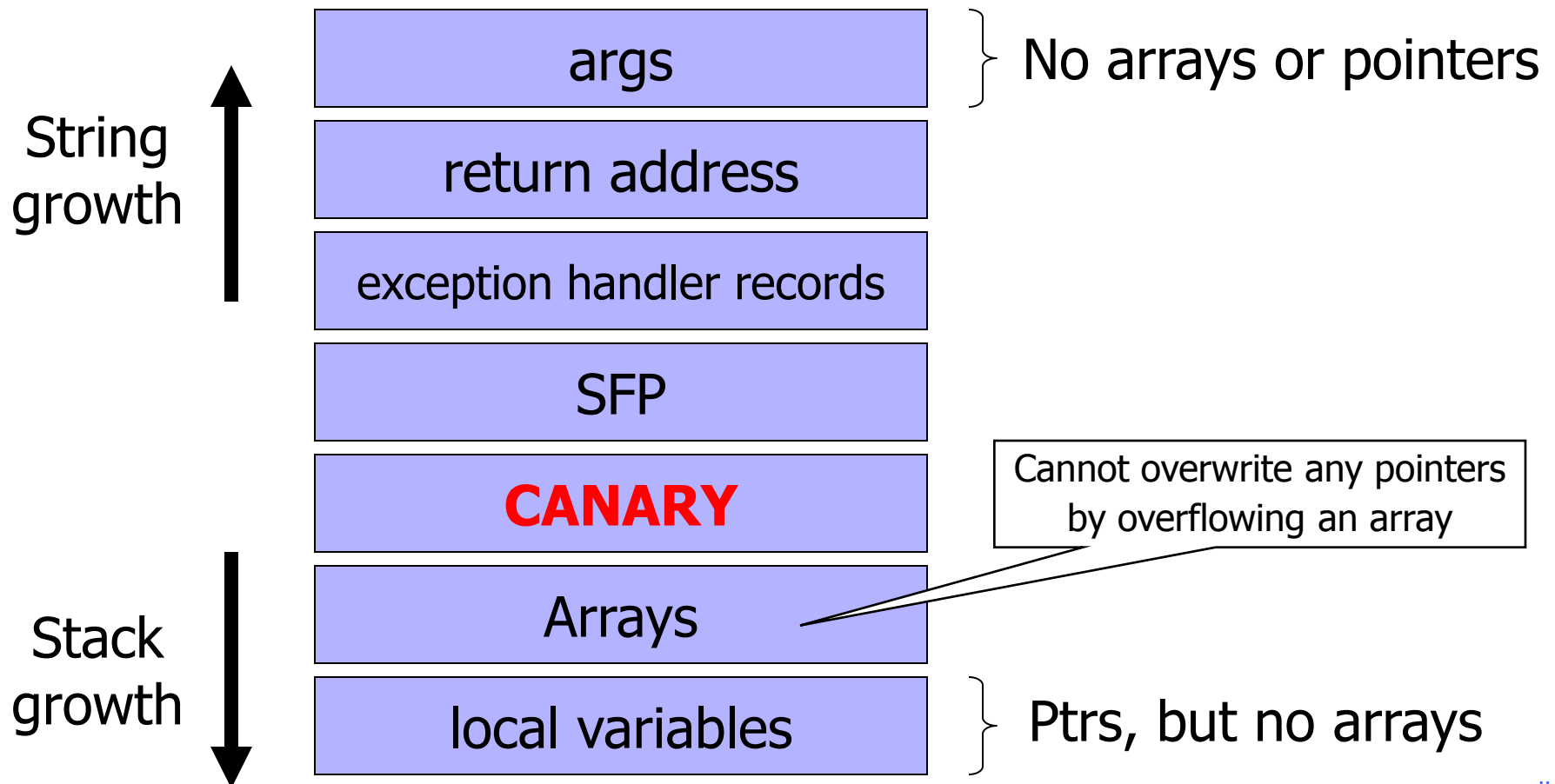
- ◆ Suppose program contains `strcpy(dst,buf)` where attacker controls both `dst` and `buf`
 - Example: `dst` is a local pointer variable



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

SafeSEH: Safe Exception Handling

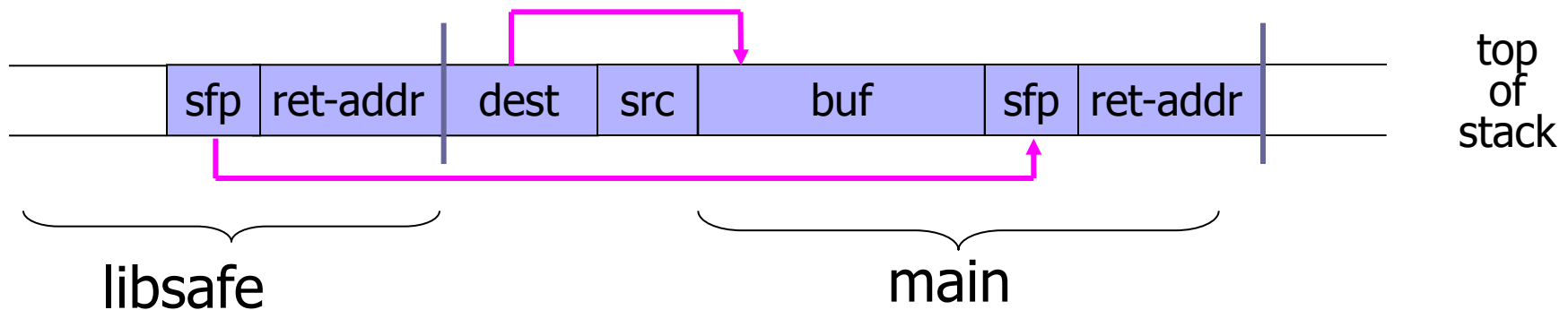
- ◆ Exception handler record must be on the stack of the current thread (why?)
- ◆ Must point outside the stack (why?)
- ◆ Must point to a valid handler
 - Microsoft's /SafeSEH linker option: header of the binary lists all valid handlers
- ◆ Exception handler records must form a linked list, terminating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

SEHOP

- ◆ SEHOP: Structured Exception Handling Overwrite Protection (since Win Vista SP1)
- ◆ Observation: SEH attacks typically corrupt the “next” entry in SEH list
- ◆ SEHOP adds a dummy record at top of SEH list
- ◆ When exception occurs, dispatcher walks up list and verifies dummy record is there; if not, terminates process

Libsafe

- ◆ Dynamically loaded library – no need to recompile!
- ◆ Intercepts calls to `strcpy(dest, src)`, other unsafe C library functions
 - Checks if there is sufficient space in current stack frame $|\text{framePointer} - \text{dest}| > \text{strlen}(\text{src})$
 - If yes, does `strcpy`; else terminates application



Limitations of Libsafe

- ◆ Protects frame pointer and return address from being overwritten by a stack overflow
- ◆ Does not prevent sensitive local variables below the buffer from being overwritten
- ◆ Does not prevent overflows on global and dynamically allocated buffers

ASLR: Address Space Randomization

- ◆ Map shared libraries to a random location in process memory
 - Attacker does not know addresses of executable code
- ◆ Deployment
 - Windows Vista: 8 bits of randomness for DLLs
 - If aligned to 64K page in a 16MB region, then 256 choices
 - Linux (via PaX): 16 bits of randomness for libraries
 - More effective on 64-bit architectures
- ◆ Other randomization methods
 - Randomize system call ids or instruction set

Example: ASLR in Vista

Booting Vista twice loads libraries into different locations:

ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows

ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

ASLR is only applied to images for which
the **dynamic-relocation** flag is set

Other Targets of Memory Exploits

- ◆ 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
- ◆ None of these exploits violate the integrity of the program's control flow
 - Only original program code is executed!

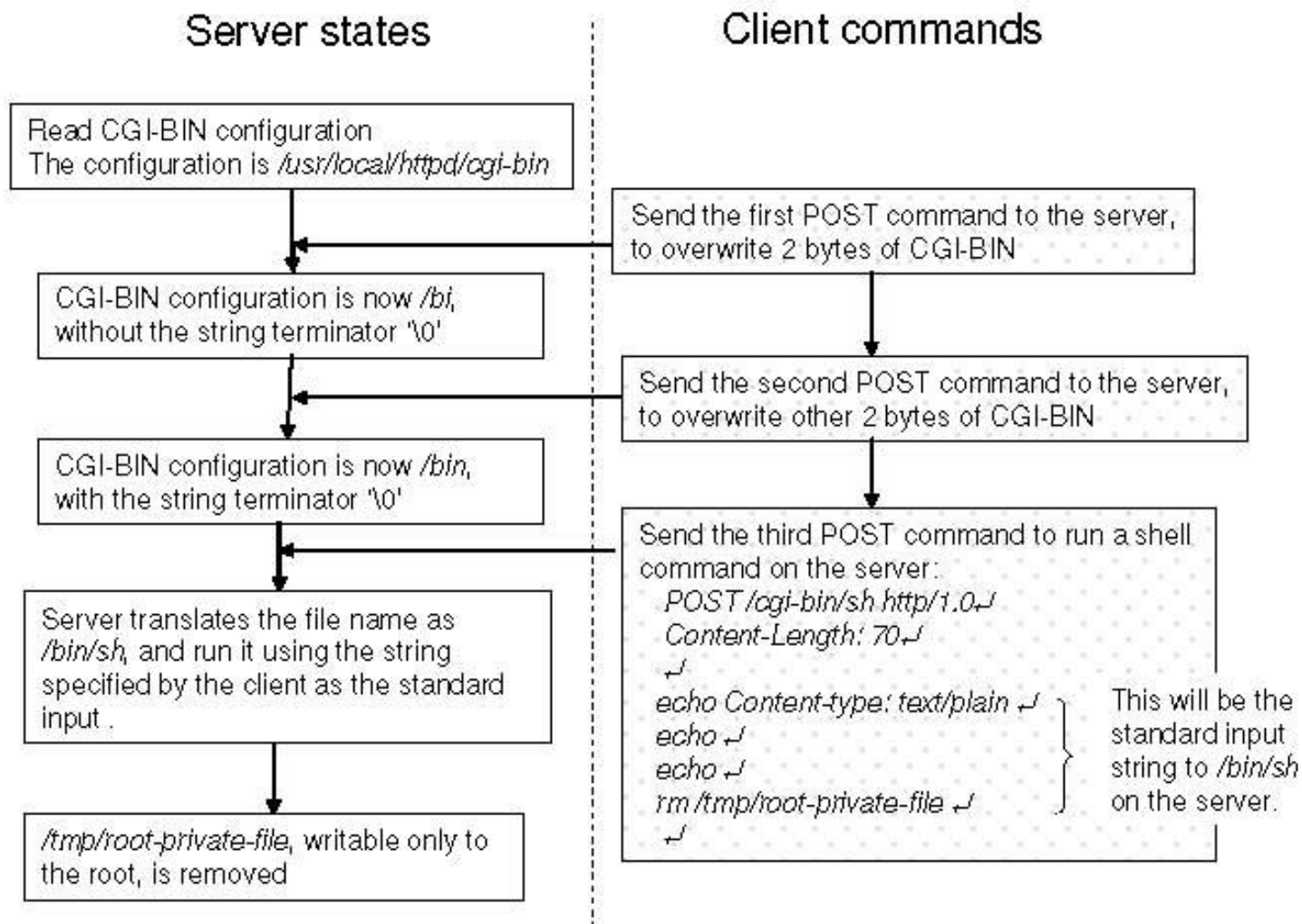
Example: Web Server Security

- ◆ **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 a corrupted buffer is freed, an overflowed value is copied to a location whose address is also read from an overflowed memory area
 - This enables the attacker to write an arbitrary value into a memory location of his choice
- ◆ Standard exploit: write 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

Check that URL doesn't contain "/.."

```
int serveconnection(int sockfd) {
    char *ptr; // pointer to the URL.
               // ESI is allocated
               // to this variable.
    ...
1: if (strstr(ptr, "/.."))
    reject the request;
2: log(...);
3: if (strstr(ptr, "cgi-bin"))
4:   Handle CGI request
    ...
}
```

Register containing pointer to URL is pushed onto stack...

```
Assembly of log(...)
push %ebp
mov %esp, %ebp
push %edi
push %esi
push %ebx
... stack buffer overflow code
pop %ebx
pop %esi
pop %edi
pop %ebp
ret
```

At this point, overflown ptr may point to a string containing "/.."

... overflown
... and read from stack

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; write 1 here  
    ...  
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);  
}
```

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

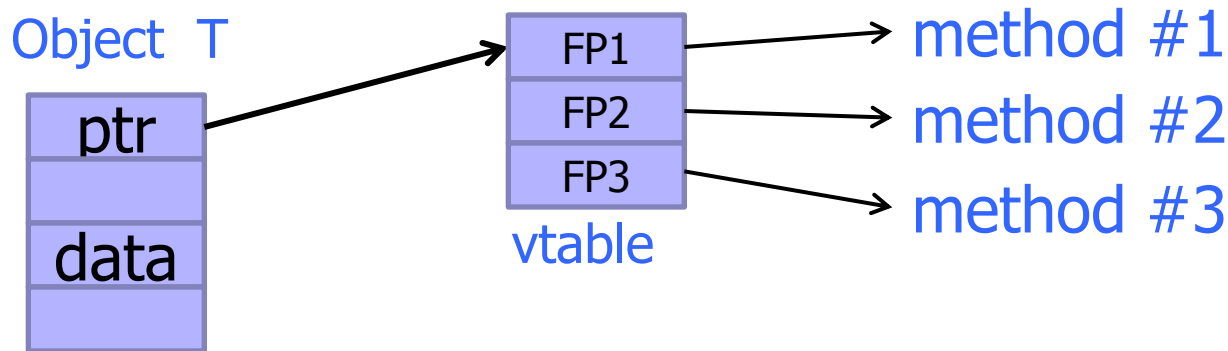
```
(B2) Modified SSHD do_authentication()  
{ int authenticated = 0;  
  while (!authenticated) {  
L1:type = packet_read(); //vulnerable  
  authenticated = 0; ——— Reset flag here, right before  
    switch (type) {                                     doing the checks  
      case SSH_CMSG_AUTH_PASSWORD:  
        if (auth_password(user, passwd))  
          authenticated = 1;  
      case ...  
    }  
    if (authenticated) break;  
  }  
  do_authenticated(pw);  
}
```

Heap Overflow

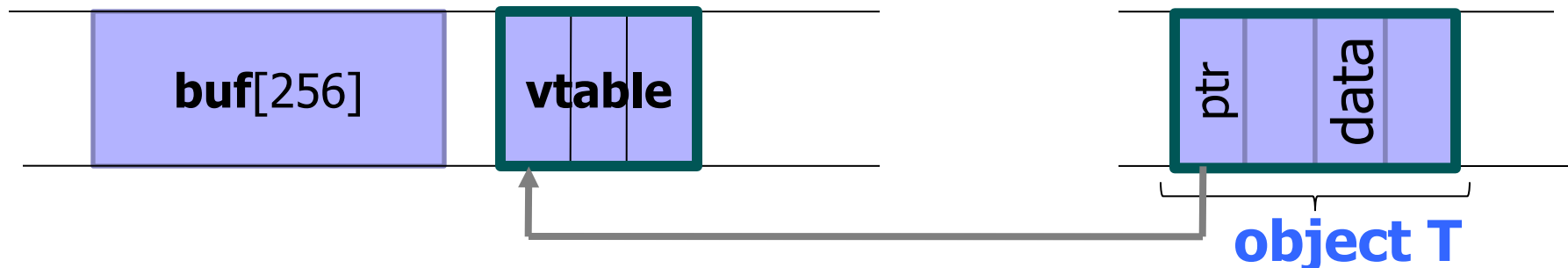
- ◆ Overflowing buffers on heap can change pointers that point to important data
 - **Illegitimate privilege elevation:** if program with overflow has sysadm/root rights, attacker can use it to write into a normally inaccessible file
 - Example: replace a filename pointer with a pointer into a memory location containing the name of a system file (for example, instead of temporary file, write into AUTOEXEC.BAT)
- ◆ Sometimes can transfer execution to attack code
 - Example: December 2008 attack on XML parser in Internet Explorer 7 - see <http://isc.sans.org/diary.html?storyid=5458>

Function Pointers on the Heap

Compiler-generated function pointers
(e.g., virtual method table in C++ or JavaScript code)

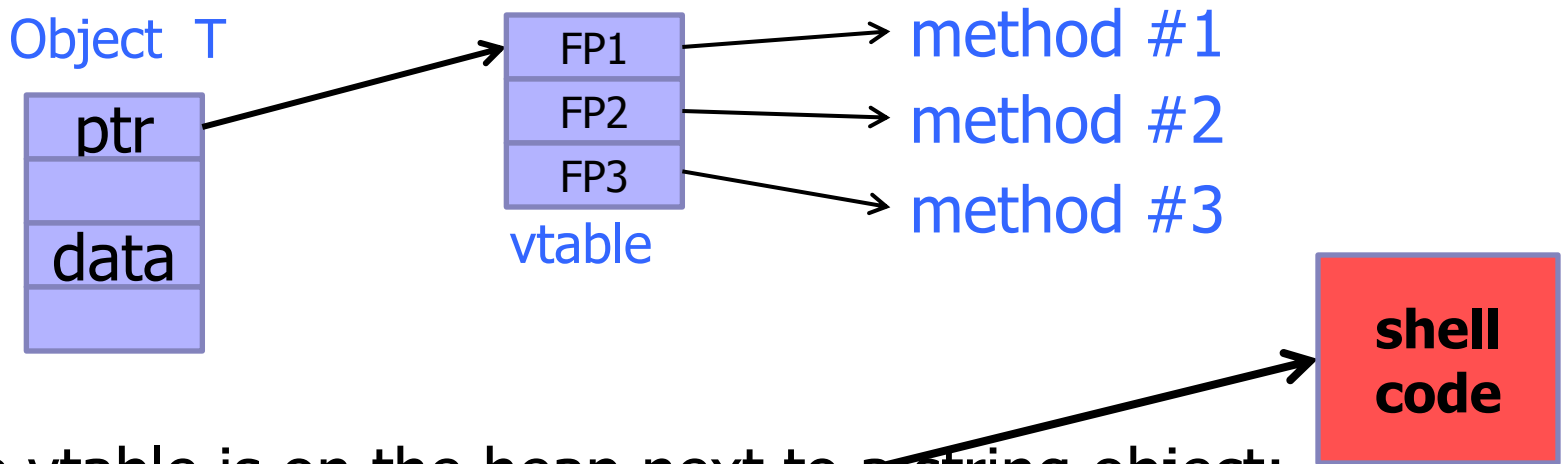


Suppose vtable is on the heap next to a string object:

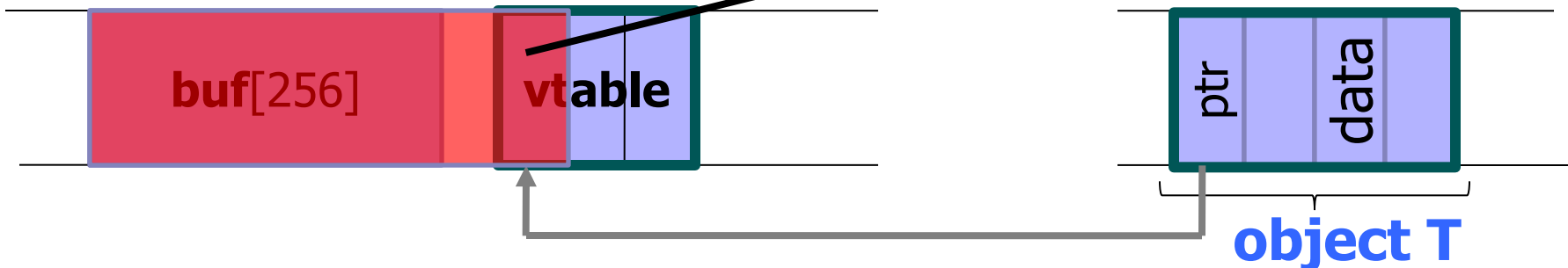


Heap-Based Control Hijacking

Compiler-generated function pointers
(e.g., virtual method table in C++ code)



Suppose vtable is on the heap next to a string object:



Problem?

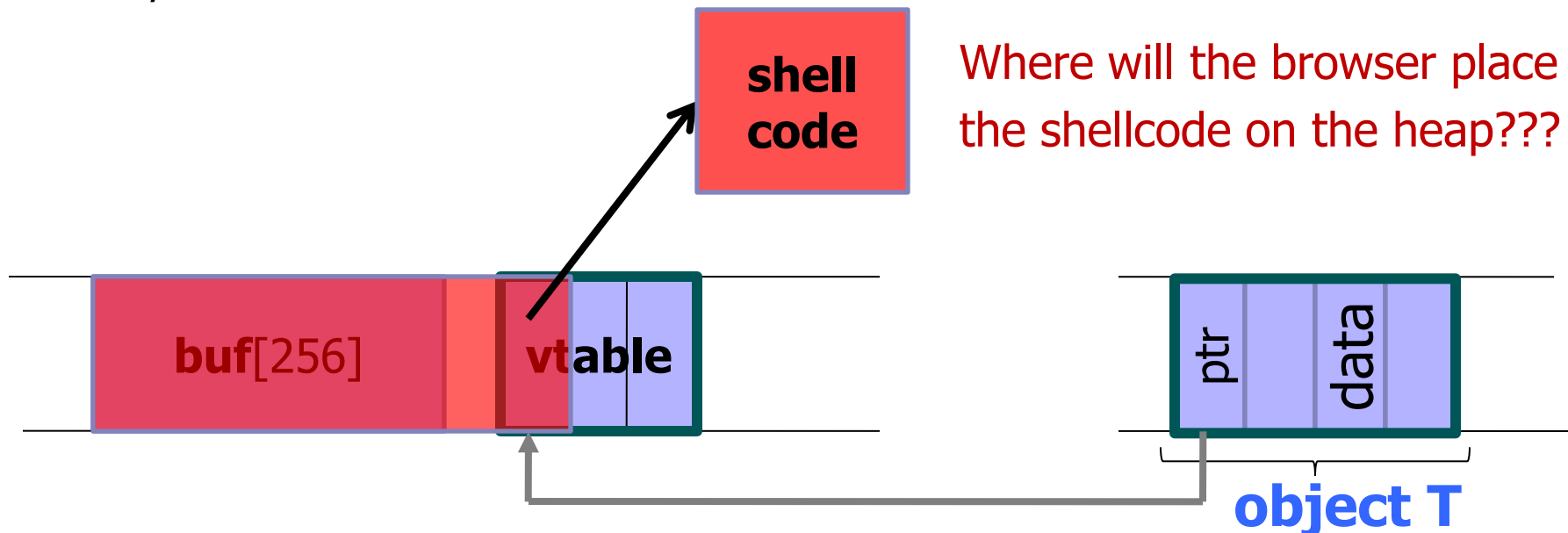
```
<SCRIPT language="text/javascript">
```

```
  shellcode = unescape("%u4343%u4343%...");
```

```
  overflow-string = unescape("%u2332%u4276%...");
```

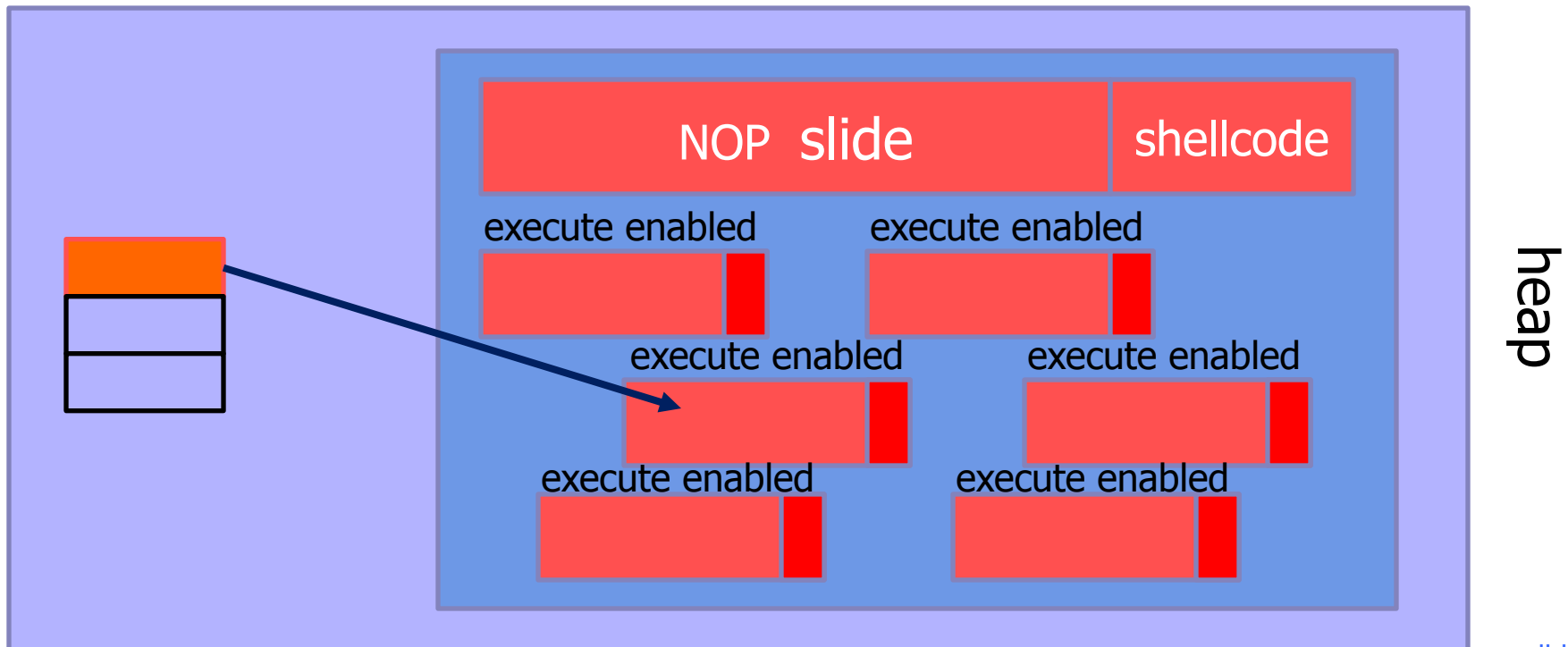
```
  cause-overflow( overflow-string );    // overflow buf[ ]
```

```
</SCRIPT?>
```



Heap Spraying

- ◆ Force JavaScript JiT (“just-in-time” compiler) to fill heap with executable shellcode, then point SFP or vtable ptr anywhere in the spray area



JavaScript Heap Spraying

```
var nop = unescape("%u9090%u9090")  
while (nop.length < 0x100000) nop += nop  
  
var shellcode = unescape("%u4343%u4343%...");
```

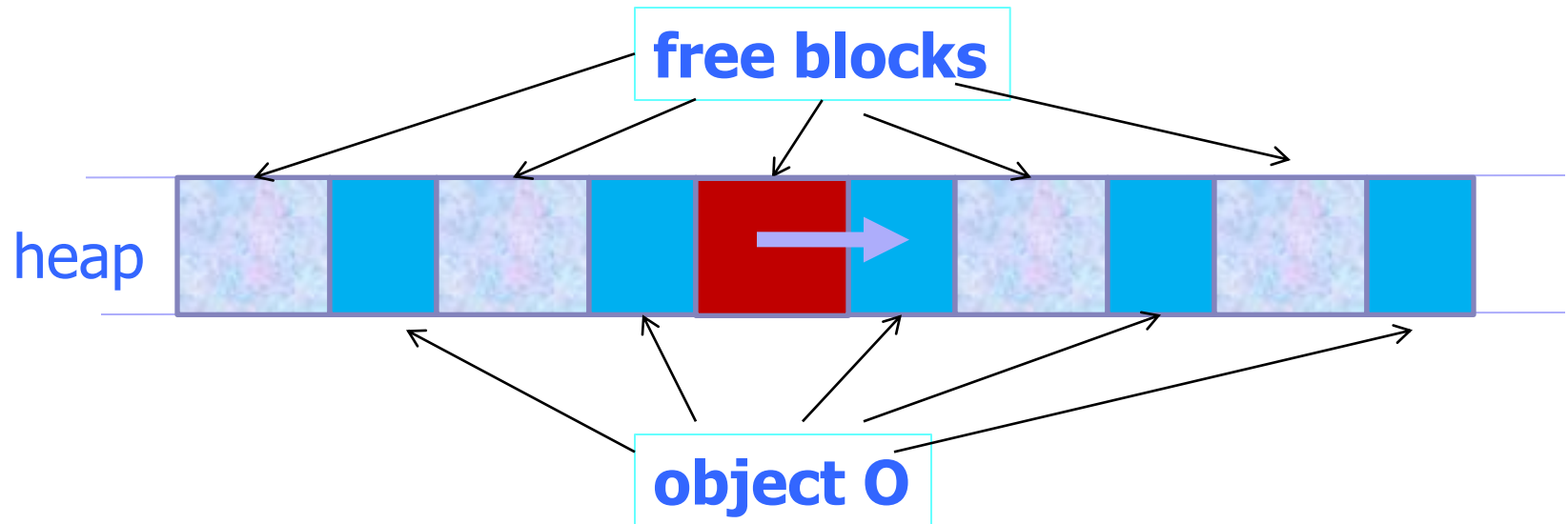
```
var x = new Array ()  
for (i=0; i<1000; i++) {  
    x[i] = nop + shellcode;  
}
```

- ◆ Pointing a function pointer anywhere in the heap will cause shellcode to execute

Placing Vulnerable Buffer

[Safari PCRE exploit, 2008]

- ◆ Use a sequence of JavaScript allocations and free's to make the heap look like this:



- ◆ Allocate vulnerable buffer in JavaScript and cause overflow

Dynamic Memory Management in C

◆ Memory allocation: `malloc(size_t n)`

- Allocates `n` bytes and returns a pointer to the allocated memory; memory not cleared
- Also `calloc()`, `realloc()`

◆ Memory deallocation: `free(void * p)`

- Frees the memory space pointed to by `p`, which must have been returned by a previous call to `malloc()`, `calloc()`, or `realloc()`
- If `free(p)` has already been called before, undefined behavior occurs
- If `p` is `NULL`, no operation is performed

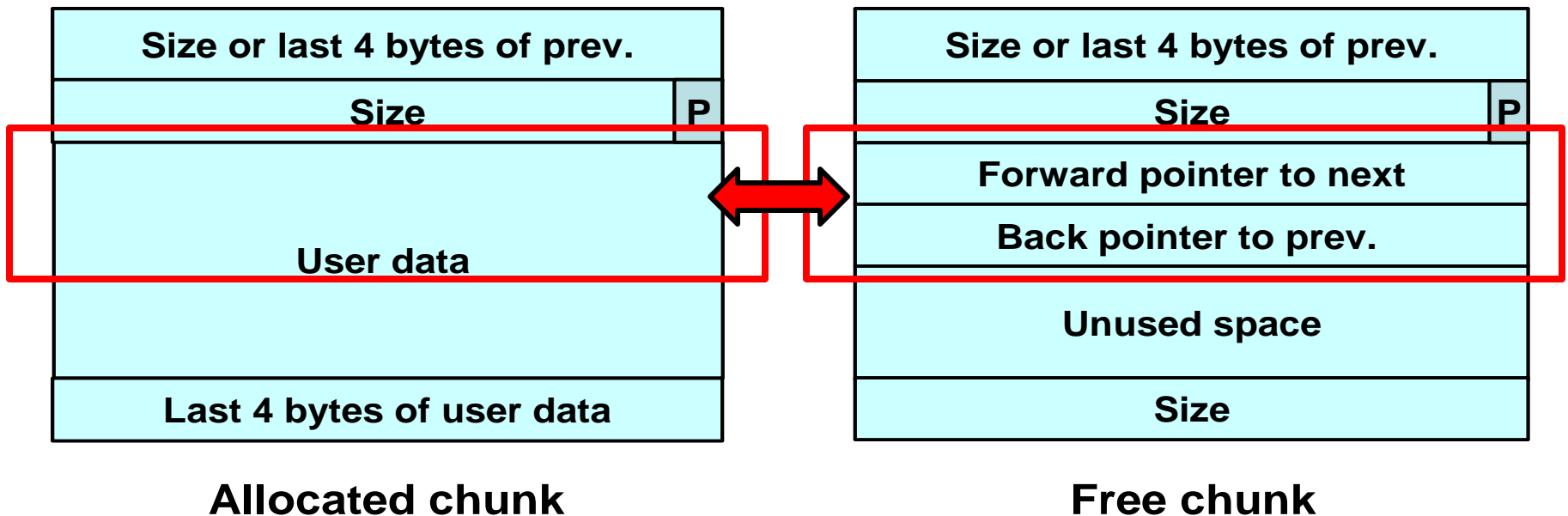
Memory Management Errors

- ◆ Initialization errors
- ◆ Failing to check return values
- ◆ Writing to already freed memory
- ◆ Freeing the same memory more than once
- ◆ Improperly paired memory management functions (example: malloc / delete)
- ◆ Failure to distinguish scalars and arrays
- ◆ Improper use of allocation functions

All result in exploitable vulnerabilities

Doug Lea's Memory Allocator

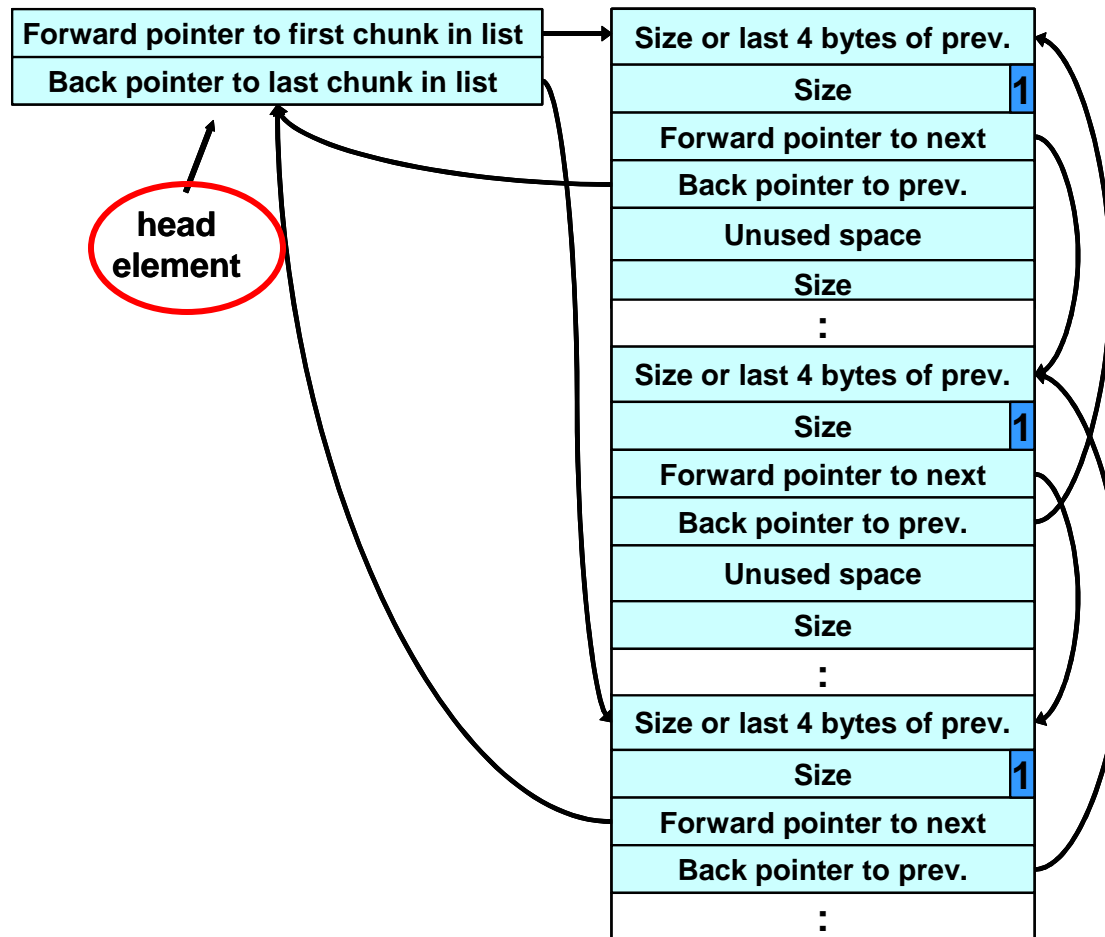
- ◆ The GNU C library and most versions of Linux are based on Doug Lea's malloc (dlmalloc) as the default native version of malloc



Free Chunks in dlmalloc

- ◆ Organized into circular double-linked lists (bins)
- ◆ Each chunk on a free list contains forward and back pointers to the next and previous chunks in the list
 - These pointers in a free chunk occupy the same eight bytes of memory as user data in an allocated chunk
- ◆ Chunk size is stored in the last four bytes of the free chunk
 - Enables adjacent free chunks to be consolidated to avoid fragmentation of memory

A List of Free Chunks in dlmalloc



Responding to Malloc

◆ Best-fit method

- An area with m bytes is selected, where m is the smallest available chunk of contiguous memory equal to or larger than n (requested allocation)

◆ First-fit method

- Returns the first chunk encountered containing n or more bytes

◆ Prevention of fragmentation

- Memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful

The Unlink Macro

What if the allocator is confused
and this chunk has actually
been allocated...

... and user data written into it?

```
#define unlink(P, BK, FD) {  
    FD = P->fd;  
    BK = P->bk;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```

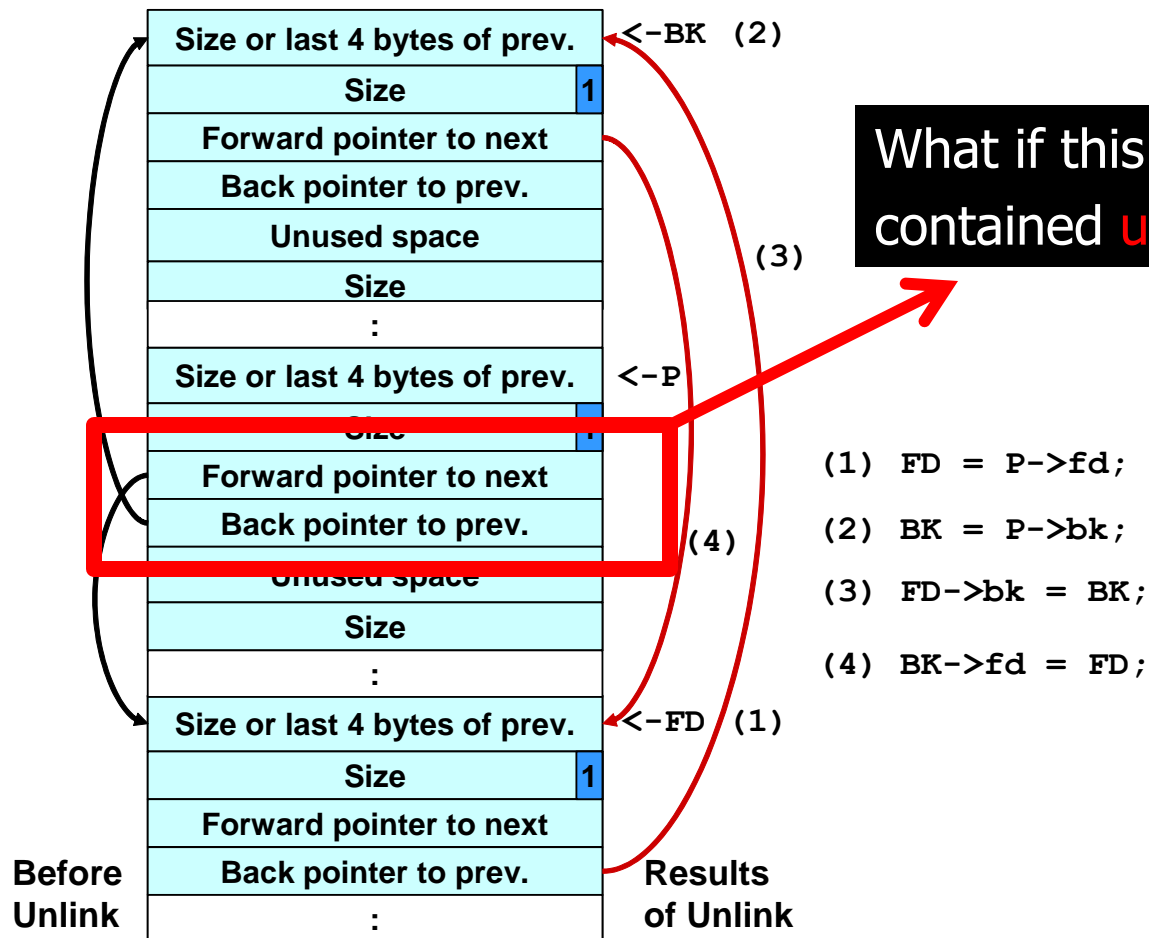
Hmm... memory copy...

Address of destination read
from the free chunk

The value to write there also read
from the free chunk

Removes a chunk from a free list -when?

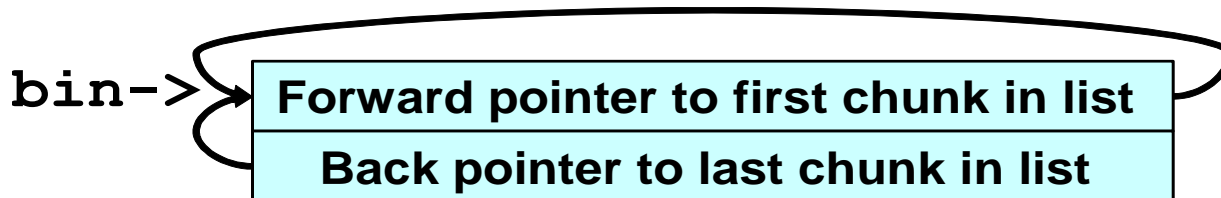
Example of Unlink



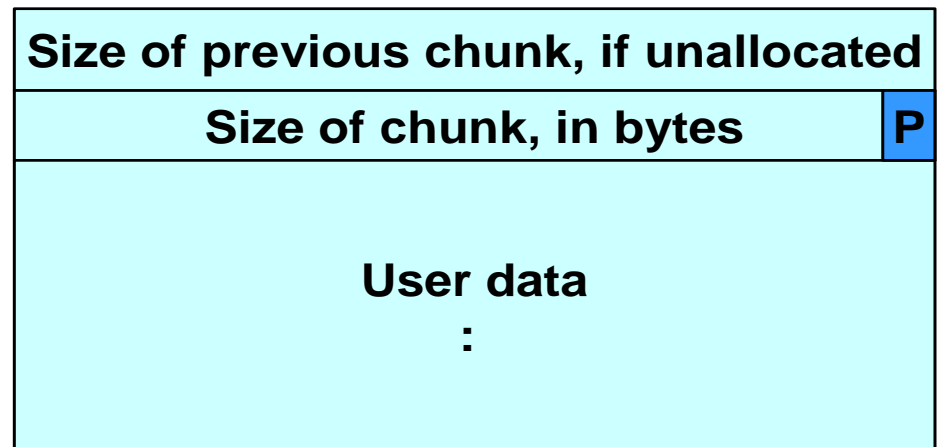
Double-Free Vulnerabilities

- ◆ Freeing the same chunk of memory twice, without it being reallocated in between
- ◆ Start with a simple case:
 - The chunk to be freed is isolated in memory
 - The bin (double-linked list) into which the chunk will be placed is empty

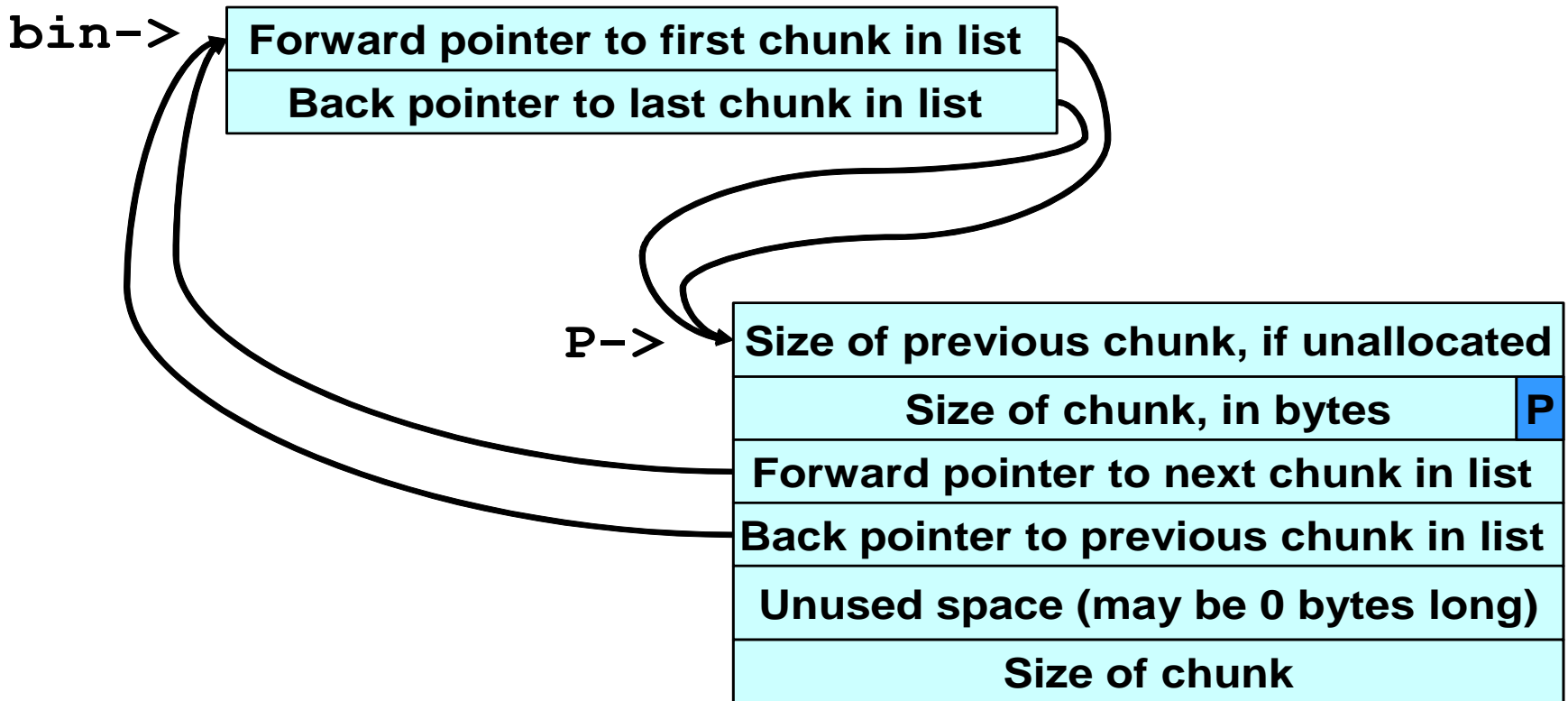
Empty Bin and Allocated Chunk



P->



After First Call to free()



After Second Call to free()

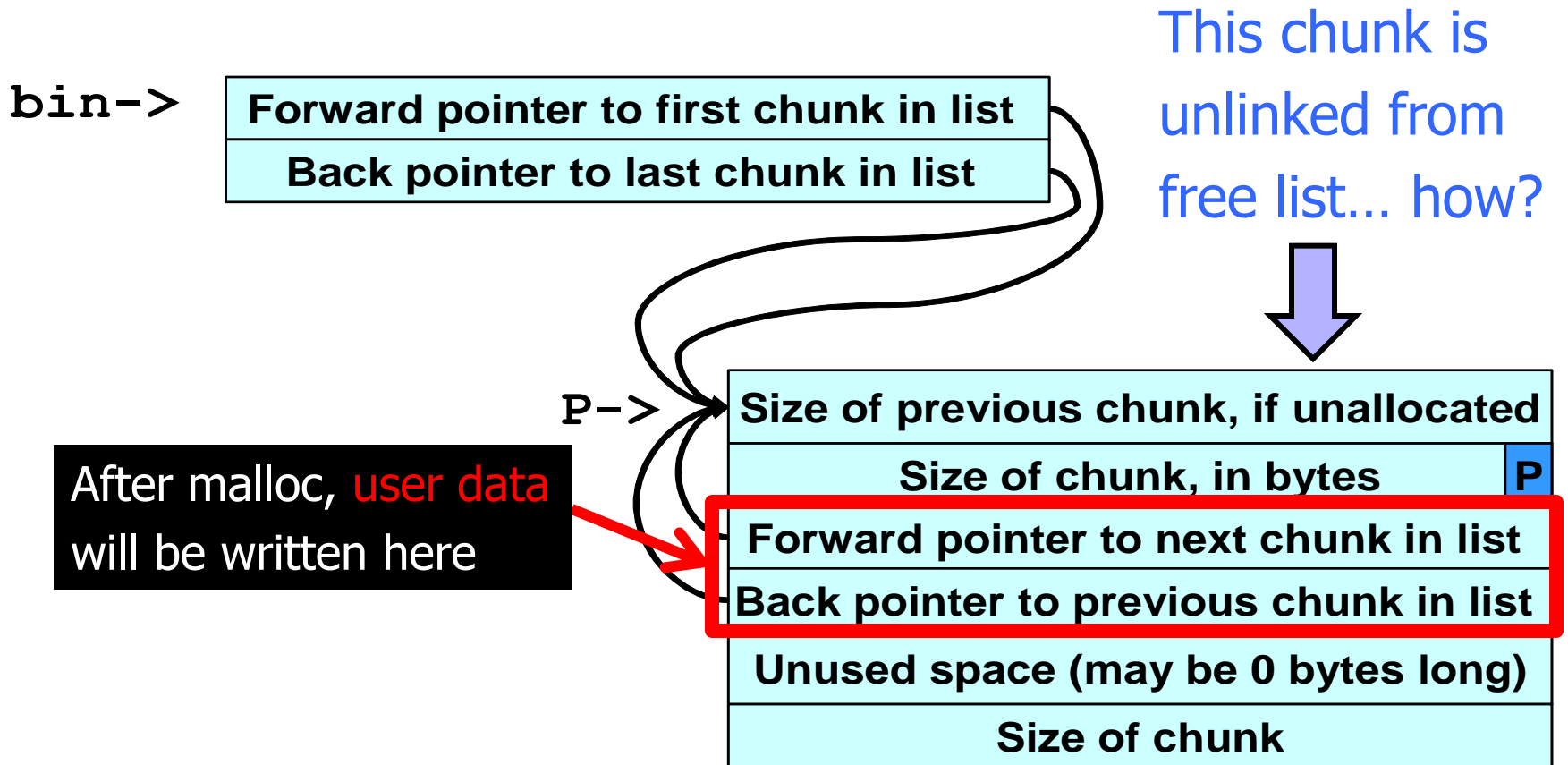
bin->

Forward pointer to first chunk in list
Back pointer to last chunk in list

P->

Size of previous chunk, if unallocated	
Size of chunk, in bytes	P
Forward pointer to next chunk in list	
Back pointer to previous chunk in list	
Unused space (may be 0 bytes long)	
Size of chunk	

After malloc() Has Been Called

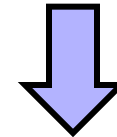


After Another malloc()

bin->

Forward pointer to first chunk in list
Back pointer to last chunk in list

Same chunk will be returned... (why?)



P->

Size of previous chunk if unallocated	
Size of chunk in bytes	P
Forward pointer to first chunk in list	
Back pointer to last chunk in list	
Unused space (in bytes long)	
Size of chunk	

After another malloc, pointers will be read from here as if it were a free chunk (why?)

One will be interpreted as address, the other as value (why?)

Sample Double-Free Exploit Code

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3. "\xeb\x0cjump12chars_"
4. "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.   int size = sizeof(shellcode);
8.   void *shellcode_location;
9.   void *first, *second, *third, *fourth;
10.  void *fifth, *sixth, *seventh;
11.  shellcode_location = (void *)malloc(size);
12.  strcpy(shellcode_location, shellcode);
13.  first = (void *)malloc(256);
14.  second = (void *)malloc(256);
15.  third = (void *)malloc(256);
16.  fourth = (void *)malloc(256);
17.  free(first);
18.  free(third);
19.  fifth = (void *)malloc(128);
20.  free(first);
21.  sixth = (void *)malloc(256);
22.  *((void **)(sixth+0))=(void *) (GOT_LOCATION-12);
23.  *((void **)(sixth+4))=(void *)shellcode_location;
24.  seventh = (void *)malloc(256);
25.  strcpy(fifth, "something");
26.  return 0;
27. }
```

First chunk free'd for the second time

This malloc returns a pointer to the same chunk as was referenced by first

The GOT address of the strcpy() function (minus 12) and the shellcode location are placed into this memory

This malloc returns same chunk yet again (why?)
unlink() macro copies the address of the shellcode into the address of the strcpy() function in the Global Offset Table - GOT (how?)

When strcpy() is called, control is transferred to shellcode... needs to jump over the first 12 bytes (overwritten by unlink)

Use-After-Free in the Real World

[ThreatPost, September 17, 2013]

The attacks are targeting IE 8 and 9 and there's no patch for the vulnerability right now... **The vulnerability exists in the way that Internet Explorer accesses an object in memory that has been deleted or has not been properly allocated.** The vulnerability may corrupt memory in a way that could allow an attacker to execute arbitrary code...

The exploit was attacking a **Use After Free vulnerability** in IE's HTML rendering engine (mshtml.dll) and was implemented entirely in Javascript (no dependencies on Java, Flash etc), but did depend on a Microsoft Office DLL which was not compiled with ASLR (Address Space Layout Randomization) enabled.

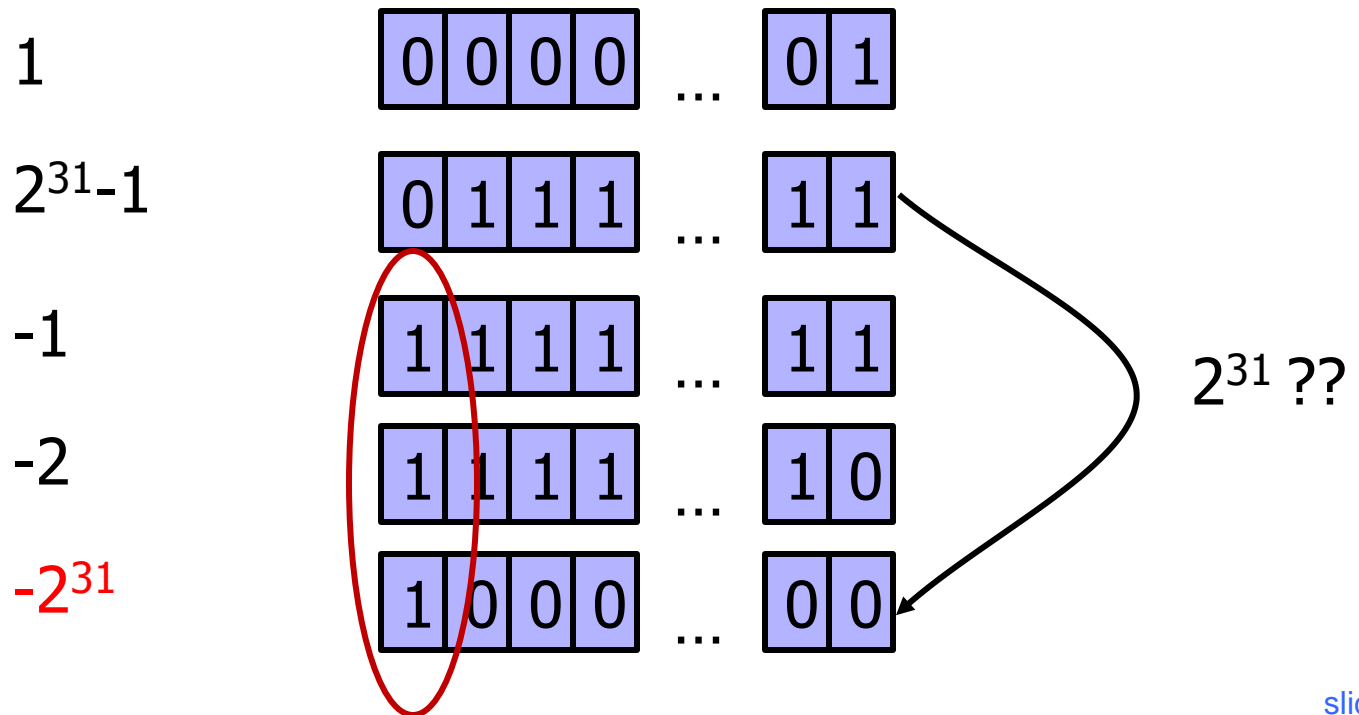
The purpose of this DLL in the context of this exploit is to bypass ASLR by providing executable code at known addresses in memory, so that a hardcoded **ROP (Return Oriented Programming)** chain can be used to mark the pages containing shellcode (in the form of Javascript strings) as executable...

The most likely attack scenarios for this vulnerability are the typical link in an email or drive-by download.

MICROSOFT WARNS OF NEW IE ZERO DAY, EXPLOIT IN THE WILD

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...

Copies "len" bytes from kernel memory to user space

... interpreted as a huge unsigned integer here

... will copy up to 4G of kernel memory

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
- ◆ Exploit published in April 2008
 - "Leveraging the ActionScript Virtual Machine"

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 short 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) { ← Invalid bytecode  
    ... 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);  
}
```

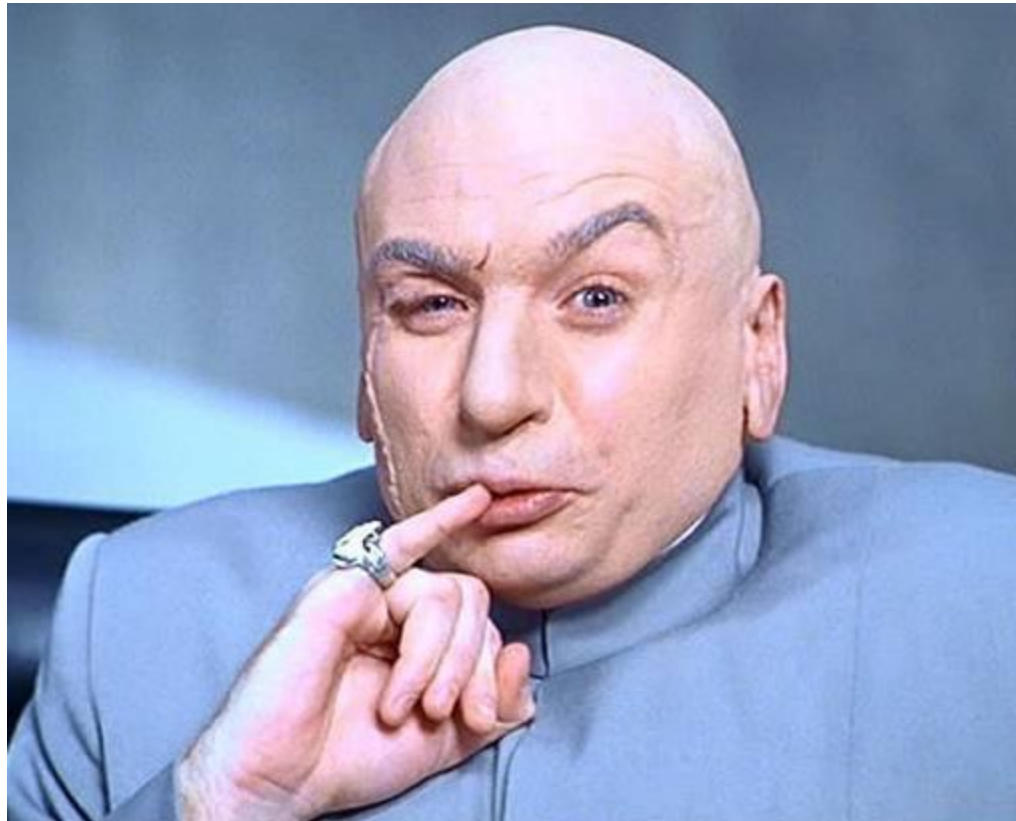
Number of operands for each opcode
is defined in AS3_argmask array

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
- ◆ 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

Variable Arguments in C

◆ In C, can define a function with a variable number of arguments

- Example: `void printf(const char* format, ...)`

◆ Examples of usage:

```
printf("hello, world");  
printf("length of %s = %d\n", str, str.length());  
printf("unable to open file descriptor %d\n", fd);
```

Format specification encoded by special % characters

- %d,%i,%o,%u,%x,%X – integer argument
- %s – string argument
- %p – pointer argument (void *)
- Several others

Implementation of Variable Args

◆ Special functions `va_start`, `va_arg`, `va_end` compute arguments at run-time

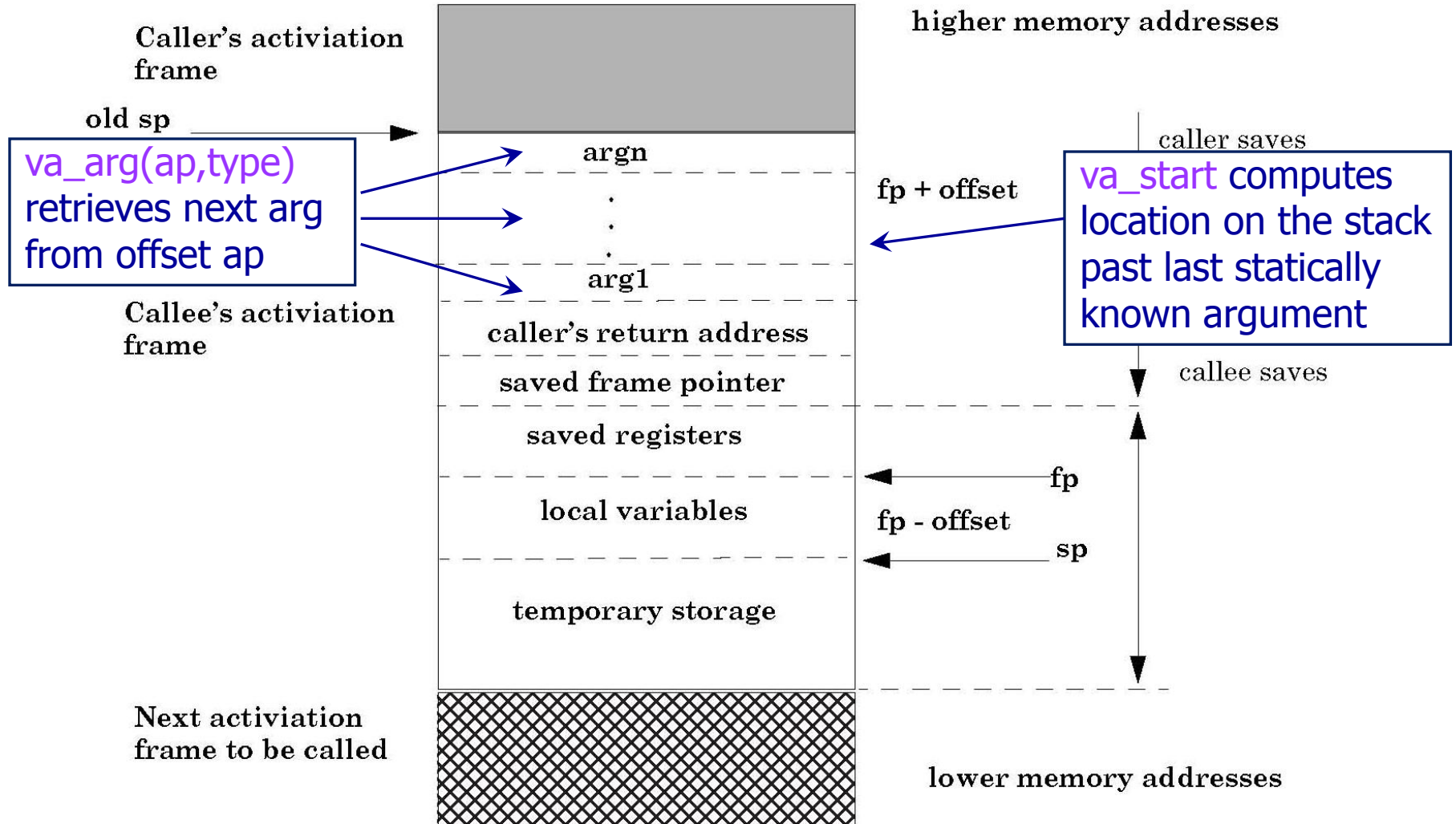
```
void printf(const char* format, ...)
{
    int i; char c; char* s; double d;
    va_list ap; /* declare an "argument pointer" to a variable arg list */
    va_start(ap, format); /* initialize arg pointer using last known arg */

    for (char* p = format; *p != '\\0'; p++) {
        if (*p == '%') {
            switch (*++p) {
                case 'd':
                    i = va_arg(ap, int); break;
                case 's':
                    s = va_arg(ap, char*); break;
                case 'c':
                    c = va_arg(ap, char); break;
            }
            ... /* etc. for each % specification */
        }
    }
    ...

    va_end(ap); /* restore any special stack manipulations */
}
```

printf has an internal stack pointer

Frame with Variable Args



Format Strings in C

◆ Proper use of printf format string:

```
... int foo=1234;  
    printf("foo = %d in decimal, %X in hex",foo,foo); ...
```

– This will print

foo = 1234 in decimal, 4D2 in hex

◆ Sloppy use of printf format string:

```
... char buf[13]="Hello, world!";  
    printf(buf);  
    // should've used printf("%s", buf); ...
```

– If the buffer contains a format symbol starting with %, location pointed to by printf's internal stack pointer will be interpreted as an argument of printf. This can be exploited to **move printf's internal stack pointer!** (how?)

Writing Stack with Format Strings

- ◆ `%n` format symbol tells `printf` to write the number of characters that have been printed

```
... printf("Overflow this!%n", &myVar); ...
```

- Argument of `printf` is interpreted as destination address
- This writes 14 into `myVar` ("Overflow this!" has 14 characters)

- ◆ What if `printf` does not have an argument?

```
... char buf[16]="Overflow this!%n";  
printf(buf); ...
```

- Stack location pointed to by `printf`'s internal stack pointer will be interpreted as address into which the number of characters will be written!

Using %n to Mung Return Address

This portion contains enough % symbols to advance printf's internal stack pointer

Buffer with attacker-supplied input string

"... attackString%n", attack code

&RET

RET

Number of characters in attackString must be equal to ... what?

Overwrite location under printf's stack pointer with RET address; printf(buffer) will write the number of characters in attackString into RET

Return execution to this address

C has a concise way of printing multiple symbols: `%Mx` will print exactly 4M bytes (taking them from the stack). Attack string should contain enough `"%Mx"` so that the number of characters printed is equal to the most significant byte of the address of the attack code.

Repeat three times (four `"%n"` in total) to write into `&RET+1`, `&RET+2`, `&RET+3`, thus replacing RET with the address of attack code byte by byte.

◆ See "Exploiting Format String Vulnerabilities" for details