Memory corruption defenses

Lecture 11

Secure Programming

Module evaluation form

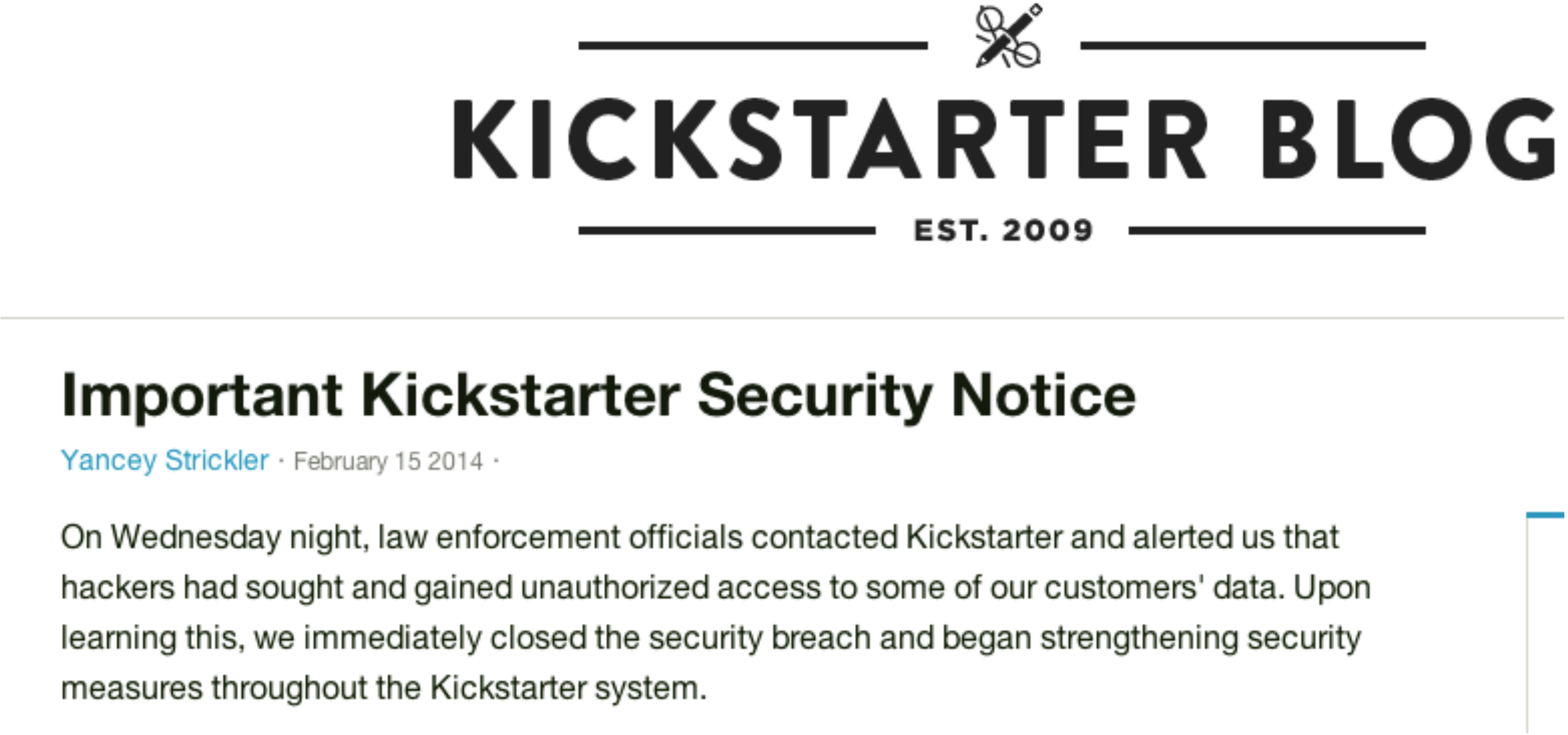
- Please take a few minutes to complete the questionnaire
- These are important!
  - Your chance to influence the module design and execution (through official channels; you can always talk to me unofficially)
  - Responses are read and taken very seriously by the whole School
- “WWW support” includes course web page, Canvas, screencasts, etc.

Rescheduling

- Next week: no lecture on Monday

- Next week: can we move Tuesday’s lecture to 9am?
- If not, can we move it to the following Monday at 11am?

In the news



KICKSTARTER BLOG

EST. 2009

Important Kickstarter Security Notice

Yancey Strickler · February 15 2014 ·

On Wednesday night, law enforcement officials contacted Kickstarter and alerted us that hackers had sought and gained unauthorized access to some of our customers' data. Upon learning this, we immediately closed the security breach and began strengthening security measures throughout the Kickstarter system.

https://www.kickstarter.com/blog/important-kickstarter-security-notice

And return-into-libc

NON EXECUTABLE STACK

Where are we?

- We have seen that a mitigation mechanism against stack-based overflow consists of marking the stack region as non executable
- As attackers, can we bypass this mechanism?

Bypassing non executable stack

- Non executable stack prevents us to execute code that we injected
- We still have the capabilities of
  - Modifying the stack arbitrarily
  - Jumping and executing to existing program locations that are marked as executable
- Can we combine these capabilities to execute arbitrary (or at least useful) code?

Bypassing non executable stack

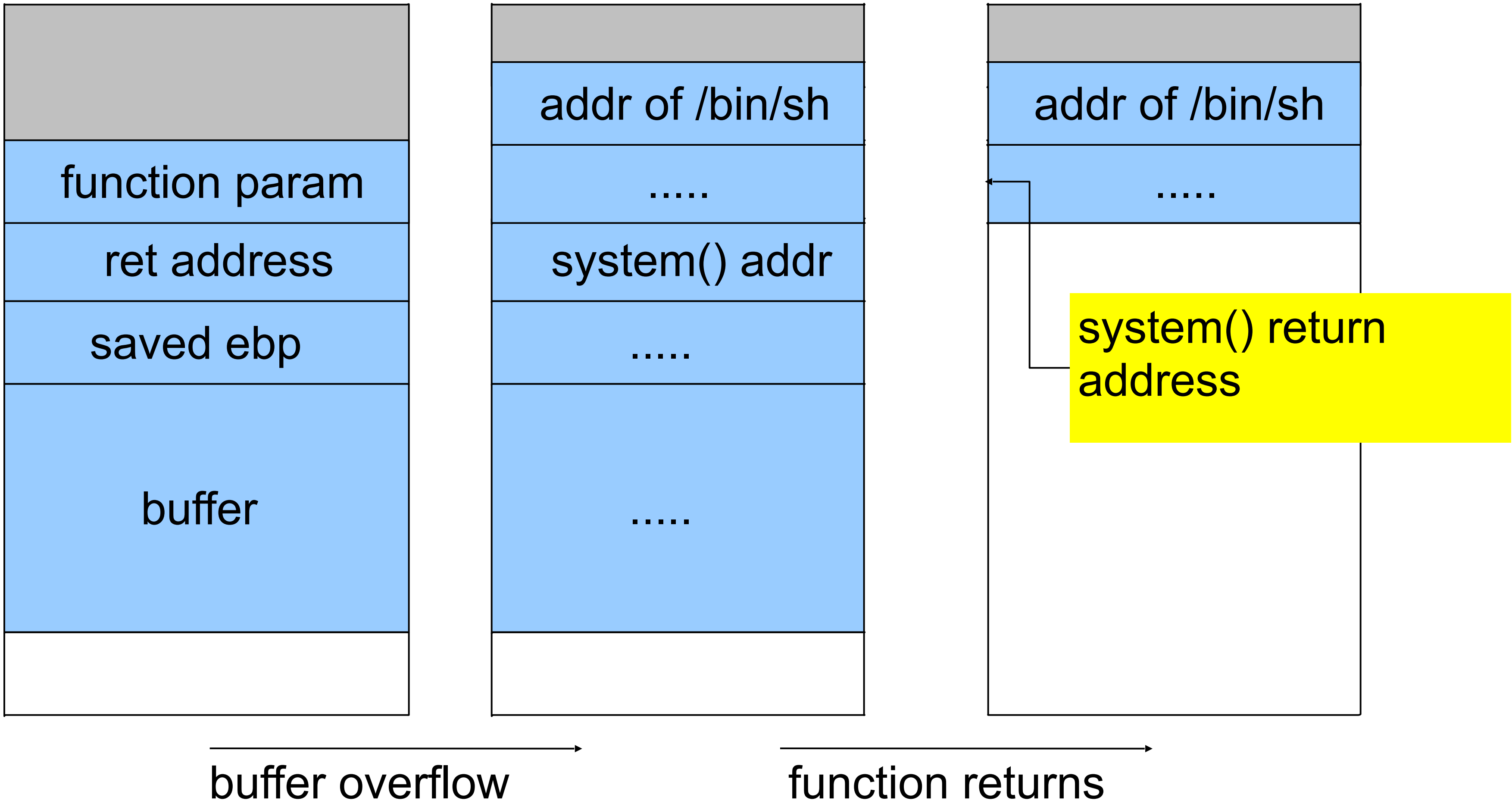
- Idea: call existing code, for example, library functions
- Attractive target is libc
  - Linked by practically all programs
  - Lots of interesting functions , e.g., `system()`, `exec*`
- Nergal, The advanced return-into-lib(c) exploits: PaX case study, 2001 http://www.phrack.org/issues.html?issue=58&id=4

Return-into-libc

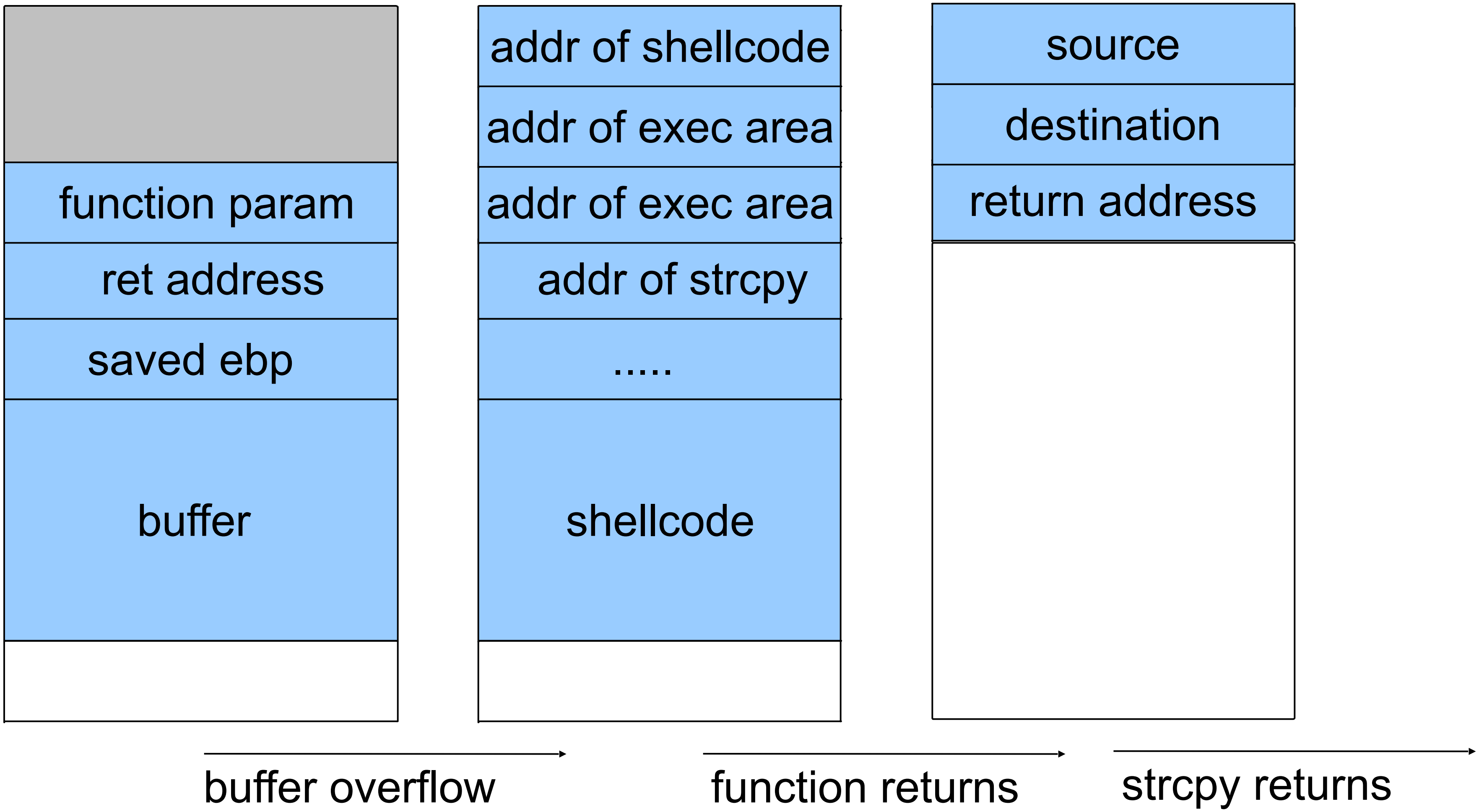
- Goal: we want to execute `system(“/bin/sh”)`
- Idea: overwrite the saved return address with the address of `system` function
- When the vulnerable function returns, `system` will start executing
- And it will look up in the stack its parameters

→ we need to set up the stack correctly

Return-into-libc: system



Return-into-libc: move shellcode



function param

ret address

saved ebp

buffer

addr of shellcode

addr of exec area

addr of exec area

addr of strcpy

.....

shellcode

source

destination

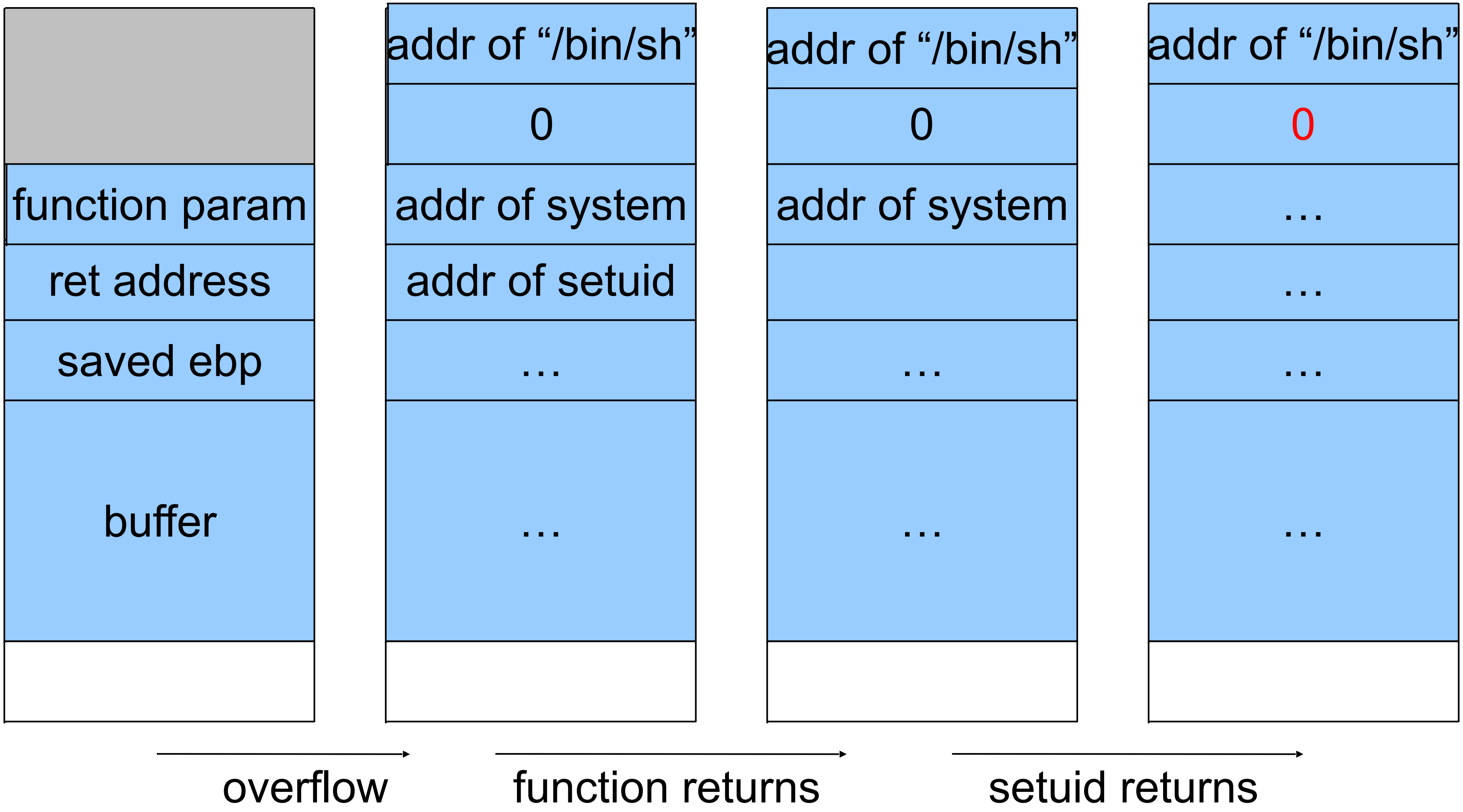
return address

buffer overflow

function returns

strcpy returns

Chaining function calls



	addr of “/bin/sh”	addr of “/bin/sh”	addr of “/bin/sh”
	0	0	0
function param	addr of system	addr of system	…
ret address	addr of setuid		…
saved ebp	…	…	…
buffer	…	…	…

overflow

function returns

setuid returns

Chaining function calls

- This technique has limitations
  - Only works for calling 2 functions
  - The first function has exactly one parameter
- Can we do better?
  - We need to find ways to control the esp

esp lifting

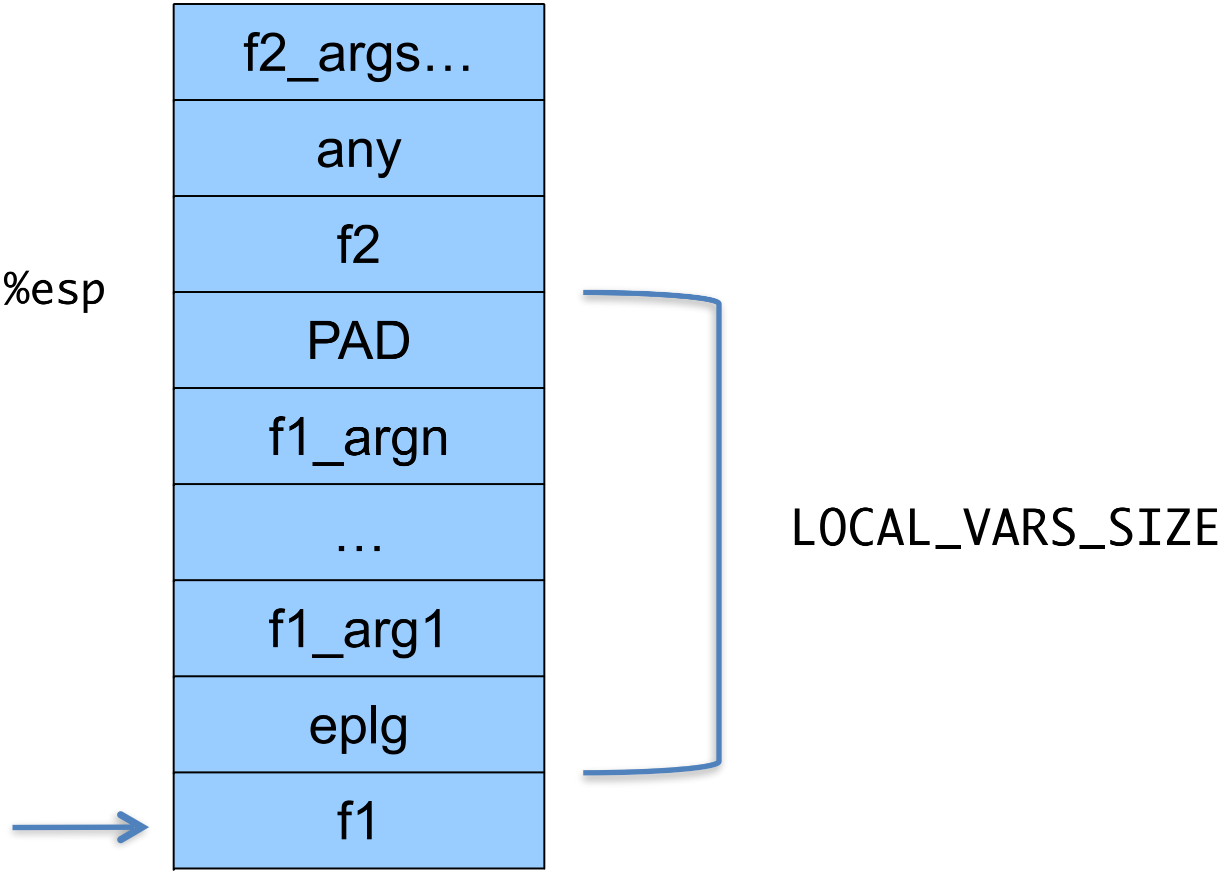
- Observation: possible to find sequences of instructions such as
  - eplg: *add* `$LOCAL_VARS_SIZE,%esp`

    ```
    ret
    ```
  - Typical function epilogue when code is compiled with -fomit-frame-pointer
- Enables us to mov $esp of fixed amount

esp lifting

- Goal: execute f1 and f2 in libc

```
eplg:
 add $LOCAL_VARS_SIZE,%esp
 ret
```



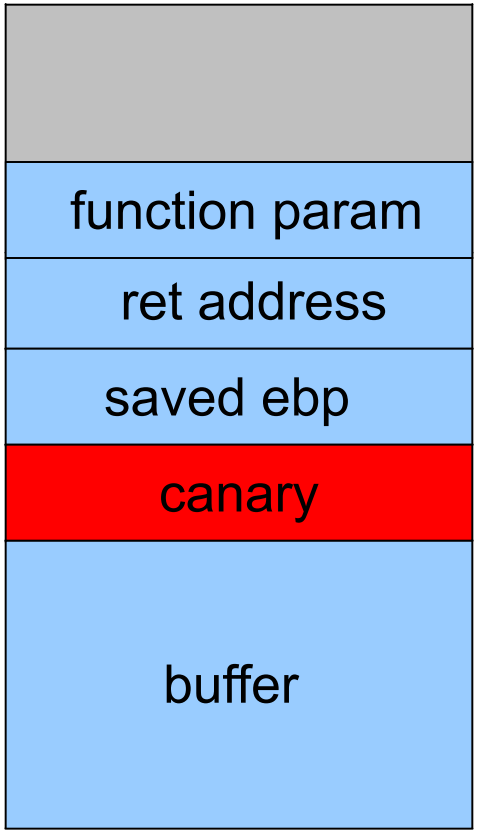
overwritten RET address

STACK PROTECTION

Stack protection

- Idea: detect whether the function frame has been overwritten as part of an attack
- Implementation: compiler
  - adds a “canary” between local variables and the saved EBP
  - at the end of the function adds a check that the canary is still alive
  - If canary value != original value, then there was an overflow

Stack protection



What canary values?

- Suggestions?

Canary values

- Terminator canaries
  - Contain string terminator characters
  - 0x00 0x00 0x0a 0xff
- Random canaries
  - Random value generated at program initialization
- Random XOR canaries
  - Random value to be XORed with data to be protected

Linux implementation

```
mov %gs:0x14,%eax
mov %eax,-0xc(%ebp)
xor %eax,%eax
…
xor %gs:0x14,%eax
je eplg
call __stack_chk_fail
eplg:
```

- %gs:0x14 contains the canary
- Values in a few executions
  - 0x4c706c00
  - 0xf59b3c00
- If check fails, terminates with `*** stack smashing detected ***`

Initializing random variables

- Reading from `/dev/urandom` at program startup
  - Somewhat inefficient
  - Consumes entropy
- Read from pool of random data provided by kernel
  - AT_RANDOM Array in ELF Auxiliary Vector

And heap spraying

ADDRESS SPACE RANDOMIZATION

Address space randomization

- Randomize the process address space so that attacker is less likely to find address to jump to
  - Stack will be positioned at different addresses
  - libc (and other variables) will be mapped at different addresses
- Instance of general concept of *artificial diversity* as a general defense mechanism
  - Source of robustness, similar to biological systems
    http://www.cs.unm.edu/~immsec/publications/hotos-97.pdf
  - Instruction set, SQL
    http://www.cs.columbia.edu/~angelos/Papers/instructionrandomization.pdf
    http://www.cs.columbia.edu/~angelos/Papers/sqlrand.pdf

Address space randomization

Without ASLR

```
$ ./aslr
buf is at 0xbffff19c
$ ./aslr
buf is at 0xbffff19c
$ ./aslr
buf is at 0xbffff19c
```

With ASLR

```
$ ./aslr
buf is at 0xbf9a1b8c
$ ./aslr
buf is at 0xbfaba92c
$ ./aslr
buf is at 0xbf9db81c
```

`/proc/sys/kernel/randomize_va_space`

ASLR design and caveats

- 32-bit systems may have few bits to randomize
  - PaX can randomize only 16 bits of the mapped area (where libraries are)
- Beware of leaks
  - Format strings may be used to reveal system's addresses
  - Probes can be used to validate guesses
    For example, return-into-libc attacks returning into usleep
- Granularity of randomization
  - Offset location of entire library vs. individual functions

ASLR design and caveats

- Re-randomization frequency
  - At process creation
  - Randomize at each probe opportunity
- Monitoring and detecting probes
  - Catch SIGSEGV and if too many in given interval, terminate program or add delay to restart
  - Do you see problems with this?
- Read more: Shacham et al., On the Effectiveness of Address-Space Randomization, CCS 2004
  http://benpfaff.org/papers/asrandom.pdf

Bypassing ASLR

- Put on your attacker’s hat
- Let’s think how we could bypass ASLR

Bypassing ASLR (1)

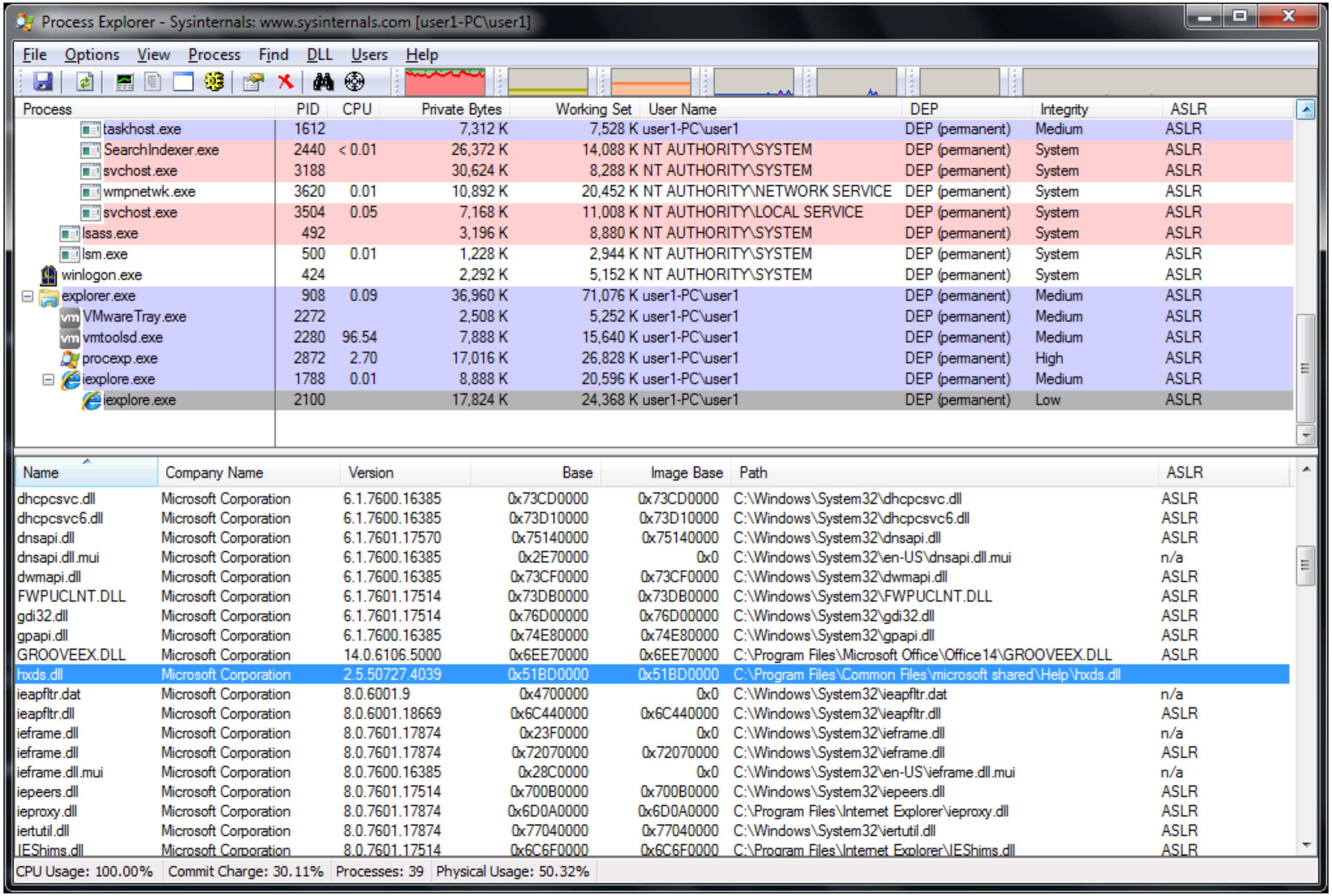
- Are all libraries on the system compiled to support ASRL?
- If not, can you (as an attacker) cause the non-ASLR library to be loaded in the address space of the vulnerable process?
- Defense is as strong as the weakest element of the defense system

ms-help ASLR bypass

- Scenario: you have found a vulnerability in IE on Windows 7
- Now you need to find (and load) a non ASLR DLL, so you can perform your ROP-based exploit
- Scan all DLLs on a “typical” system to identify those that are non ASLR
- You can load libraries via ActiveX (but this triggers a confirmation prompt from IE
- Alternative: libraries providing support for special protocol handlers, e.g., ms-help://

http://www.greyhathacker.net/?p=585

ms-help ASLR bypass


Process Explorer - Sysinternals: www.sysinternals.com [user1-PC\user1]
File
Options
View
Process
Find
DLL
Users
Help

Process	PID	CPU	Private Bytes	Working Set	User Name	DEP	Integrity	ASLR
taskhost.exe	1612		7,312 K	7,528 K	user1-PC\user1	DEP (permanent)	Medium	ASLR
SearchIndexer.exe	2440	< 0.01	26,372 K	14,088 K	NT AUTHORITY\SYSTEM	DEP (permanent)	System	ASLR
svchost.exe	3188		30,624 K	8,288 K	NT AUTHORITY\SYSTEM	DEP (permanent)	System	ASLR
wmpnetwk.exe	3620	0.01	10,892 K	20,452 K	NT AUTHORITY\NETWORK SERVICE	DEP (permanent)	System	ASLR
svchost.exe	3504	0.05	7,168 K	11,008 K	NT AUTHORITY\LOCAL SERVICE	DEP (permanent)	System	ASLR
lsass.exe	492		3,196 K	8,880 K	NT AUTHORITY\SYSTEM	DEP (permanent)	System	ASLR
lsm.exe	500	0.01	1,228 K	2,944 K	NT AUTHORITY\SYSTEM	DEP (permanent)	System	ASLR
winlogon.exe	424		2,292 K	5,152 K	NT AUTHORITY\SYSTEM	DEP (permanent)	System	ASLR
explorer.exe	908	0.09	36,960 K	71,076 K	user1-PC\user1	DEP (permanent)	Medium	ASLR
VMwareTray.exe	2272		2,508 K	5,252 K	user1-PC\user1	DEP (permanent)	Medium	ASLR
vmtoolsd.exe	2280	96.54	7,888 K	15,640 K	user1-PC\user1	DEP (permanent)	Medium	ASLR
procexp.exe	2872	2.70	17,016 K	26,828 K	user1-PC\user1	DEP (permanent)	High	ASLR
iexplore.exe	1788	0.01	8,888 K	20,596 K	user1-PC\user1	DEP (permanent)	Medium	ASLR
iexplore.exe	2100		17,824 K	24,368 K	user1-PC\user1	DEP (permanent)	Low	ASLR

Name	Company Name	Version	Base	Image Base	Path	ASLR
dhcpcsvc.dll	Microsoft Corporation	6.1.7600.16385	0x73CD0000	0x73CD0000	C:\Windows\System32\dhcpcsvc.dll	ASLR
dhcpcsvc6.dll	Microsoft Corporation	6.1.7600.16385	0x73D10000	0x73D10000	C:\Windows\System32\dhcpcsvc6.dll	ASLR
dnsapi.dll	Microsoft Corporation	6.1.7601.17570	0x75140000	0x75140000	C:\Windows\System32\dnsapi.dll	ASLR
dnsapi.dll.mui	Microsoft Corporation	6.1.7600.16385	0x2E70000	0x0	C:\Windows\System32\en-US\dnsapi.dll.mui	n/a
dwmapi.dll	Microsoft Corporation	6.1.7600.16385	0x73CF0000	0x73CF0000	C:\Windows\System32\dwmapi.dll	ASLR
FWPUCLNT.DLL	Microsoft Corporation	6.1.7601.17514	0x73DB0000	0x73DB0000	C:\Windows\System32\FWPUCLNT.DLL	ASLR
gdi32.dll	Microsoft Corporation	6.1.7601.17514	0x76D00000	0x76D00000	C:\Windows\System32\gdi32.dll	ASLR
gpapi.dll	Microsoft Corporation	6.1.7600.16385	0x74E80000	0x74E80000	C:\Windows\System32\gpapi.dll	ASLR
GROOVEEX.DLL	Microsoft Corporation	14.0.6106.5000	0x6EE70000	0x6EE70000	C:\Program Files\Microsoft Office\Office14\GROOVEEX.DLL	ASLR
hxds.dll	Microsoft Corporation	2.5.50727.4039	0x51BD0000	0x51BD0000	C:\Program Files\Common Files\microsoft shared\Help\hxds.dll	
ieapfltr.dat	Microsoft Corporation	8.0.6001.9	0x4700000	0x0	C:\Windows\System32\ieapfltr.dat	n/a
ieapfltr.dll	Microsoft Corporation	8.0.6001.18669	0x6C440000	0x6C440000	C:\Windows\System32\ieapfltr.dll	ASLR
ieframe.dll	Microsoft Corporation	8.0.7601.17874	0x23F0000	0x0	C:\Windows\System32\ieframe.dll	n/a
ieframe.dll	Microsoft Corporation	8.0.7601.17874	0x72070000	0x72070000	C:\Windows\System32\ieframe.dll	ASLR
ieframe.dll.mui	Microsoft Corporation	8.0.7600.16385	0x28C0000	0x0	C:\Windows\System32\en-US\ieframe.dll.mui	n/a
iepeers.dll	Microsoft Corporation	8.0.7601.17514	0x700B0000	0x700B0000	C:\Windows\System32\iepeers.dll	ASLR
ieproxy.dll	Microsoft Corporation	8.0.7601.17874	0x6D0A0000	0x6D0A0000	C:\Program Files\Internet Explorer\ieproxy.dll	ASLR
iertutil.dll	Microsoft Corporation	8.0.7601.17874	0x77040000	0x77040000	C:\Windows\System32\iertutil.dll	ASLR
IEShims.dll	Microsoft Corporation	8.0.7601.17514	0x6C6F0000	0x6C6F0000	C:\Program Files\Internet Explorer\IEShims.dll	ASLR

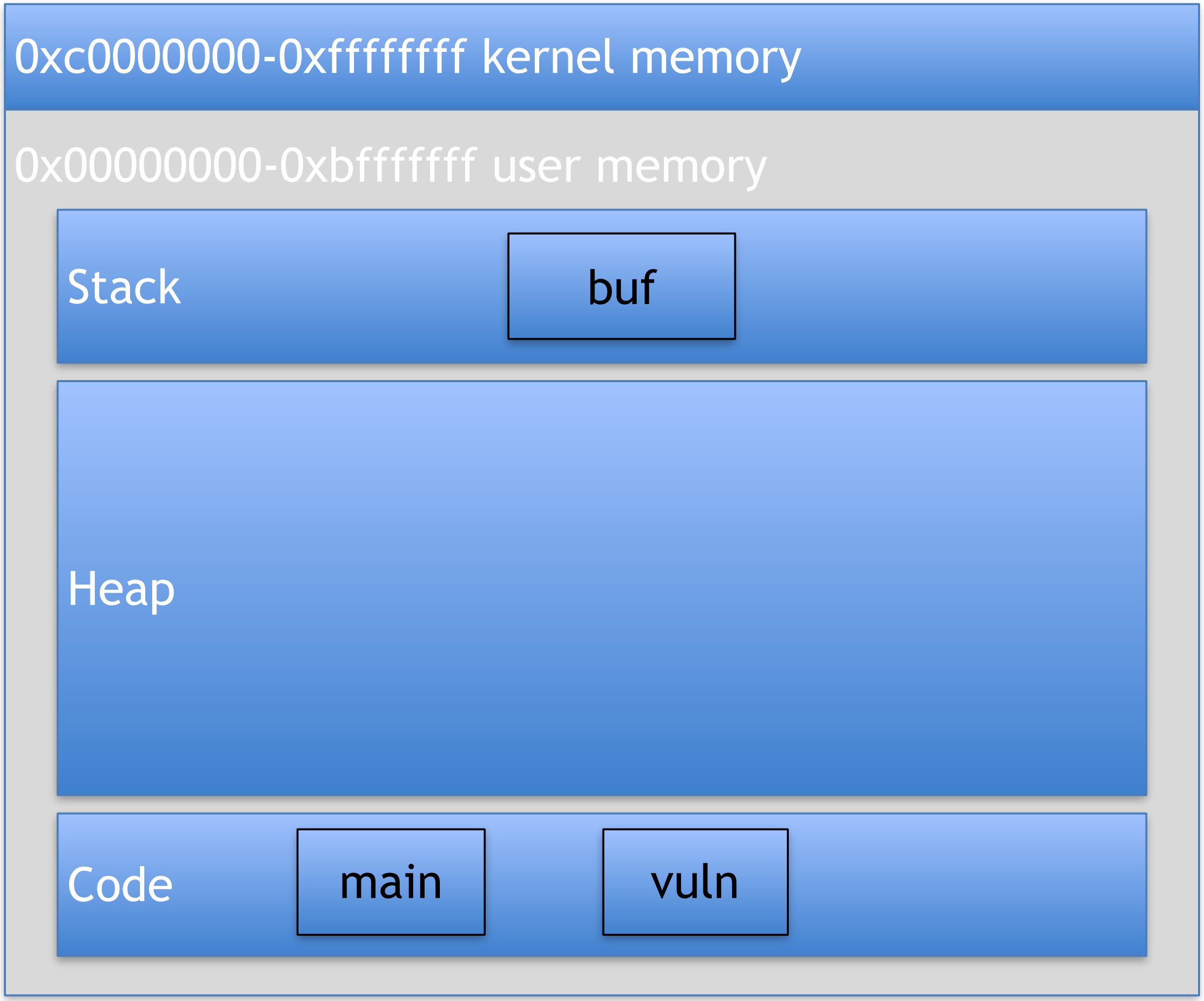
CPU Usage: 100.00% Commit Charge: 30.11% Processes: 39 Physical Usage: 50.32%

http://www.greyhathacker.net/?p=585

Bypassing ASLR (2)

- Assumption: we know how to overwrite a function pointer/return address
- But: jumping to a desired location (shellcode) is hard
- Idea: instead of trying to get the address exactly right, try to increase the chances of hitting some shellcode
  - Allocate lots of memory objects containing shellcode

Heap spraying



0xc0000000-0xffffffff kernel memory

0x00000000-0xbfffffff user memory

Stack

buf

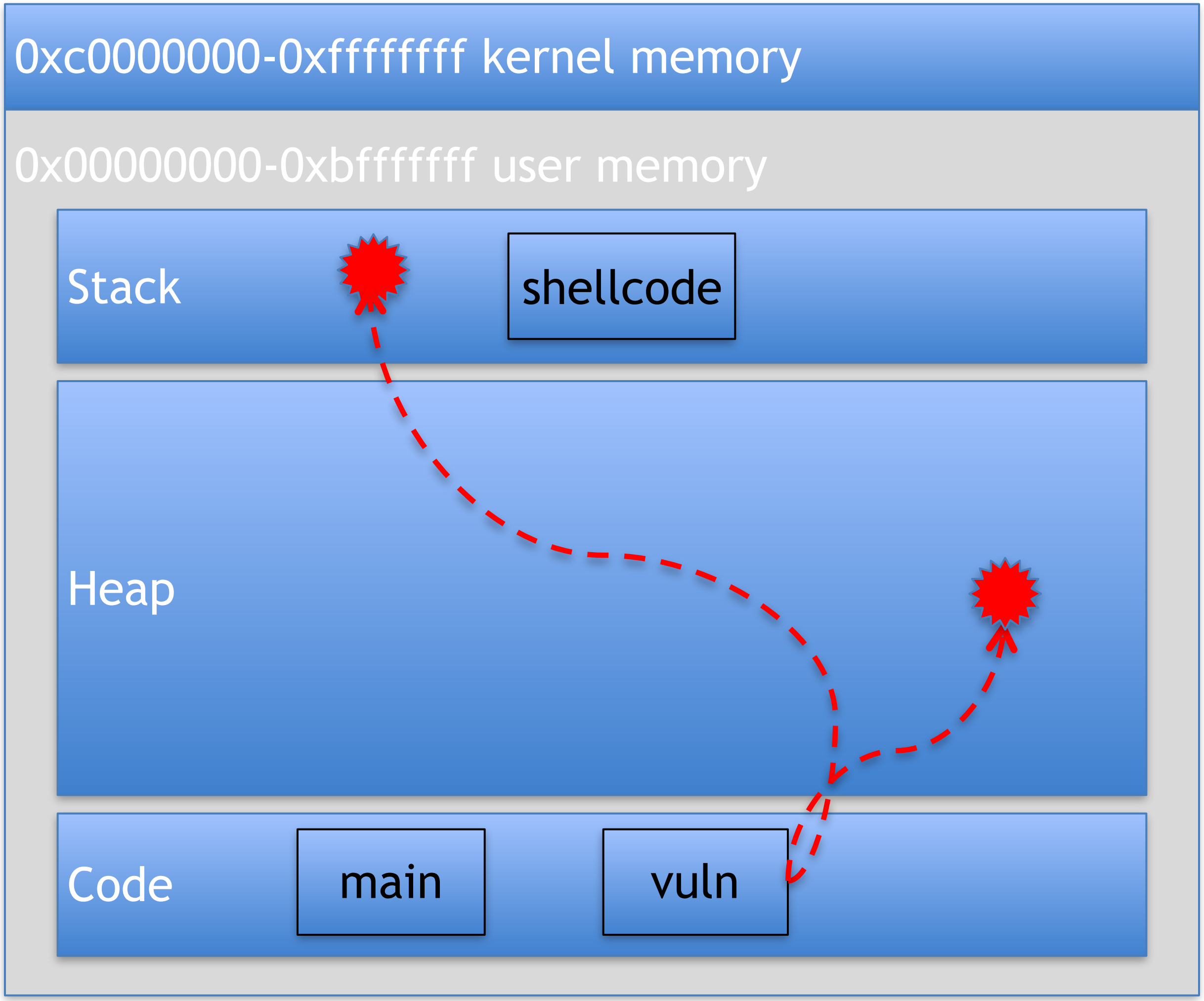
Heap

Code

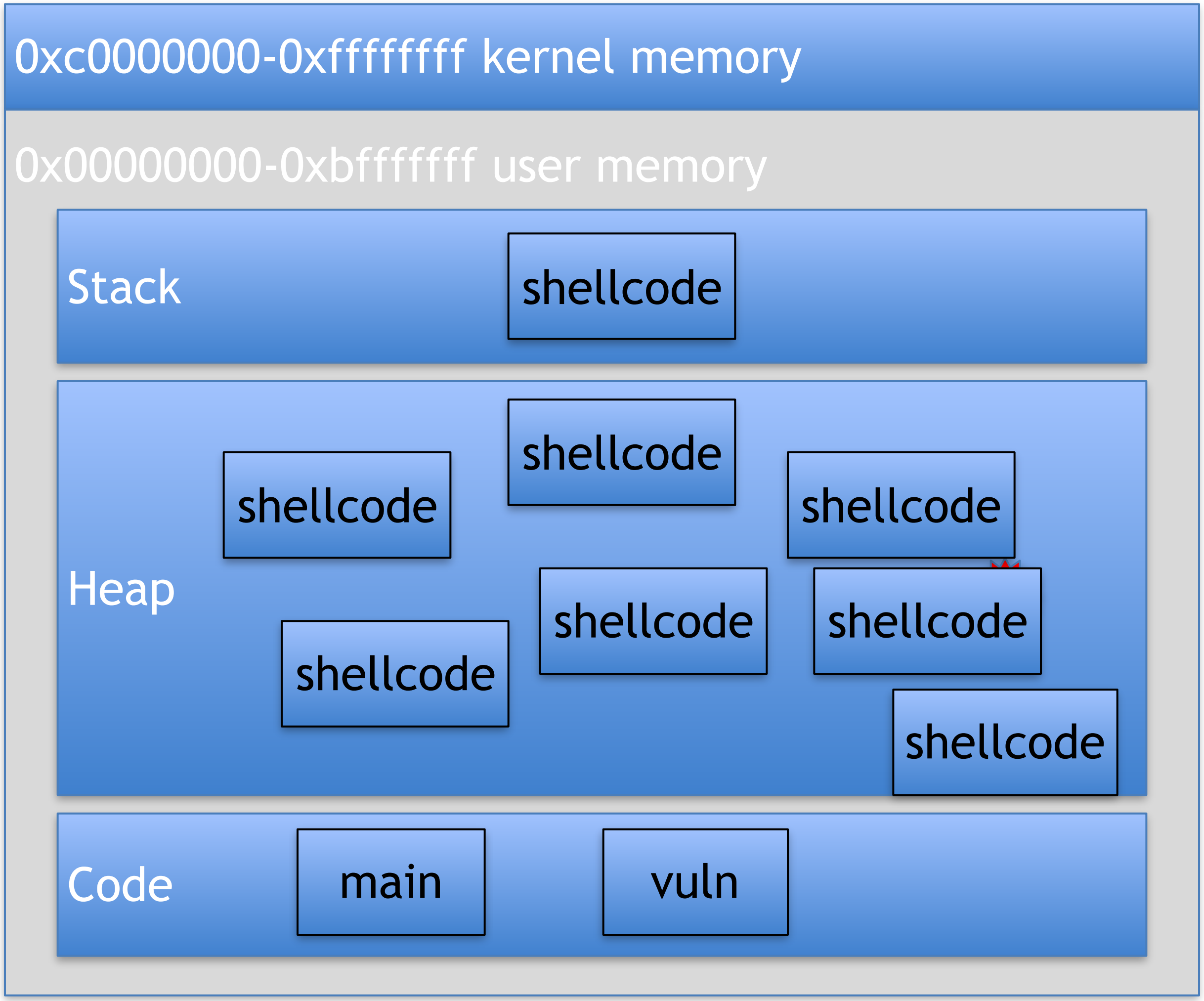
main

vuln

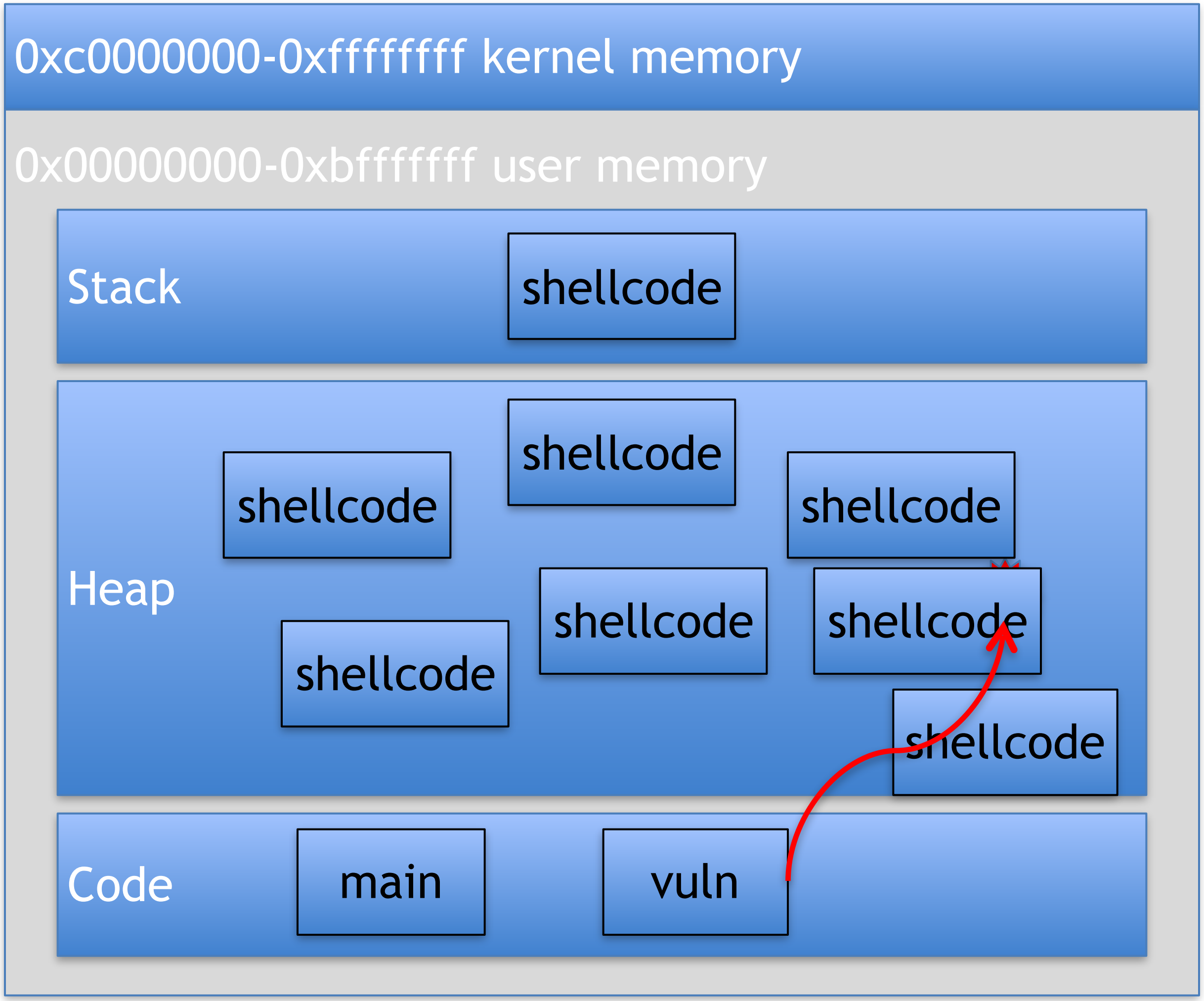
Heap spraying


0xc0000000-0xffffffff kernel memory
0x00000000-0xbfffffff user memory
Stack
shellcode
Heap
Code
main
vuln

Heap spraying



Heap spraying



0xc0000000-0xffffffff kernel memory

0x00000000-0xbfffffff user memory

Stack

shellcode

Heap

shellcode

shellcode

shellcode

shellcode

shellcode

shellcode

shellcode

Code

main

vuln

Heap spraying

- Requirements
  - Must be able to control memory allocations
  - Must be able to create many objects containing shellcode
- Easily satisfied in programs that interpret embedded scripts
  - User-provided scripts running in the context of an application
  - JavaScript (browsers, PDF readers)
  - ActionScript (Flash)

Heap spraying

- Embedded script enables attacker to allocate objects with shellcode
- They typically end up on the heap, hence “heap spraying”

```
shellcode = unescape(“..."); 


oneblock = unescape("%u0D0D%u0D0D");
var fullblock = oneblock;
while (fullblock.length<0x40000) {
    fullblock += fullblock;
}


sprayContainer = new Array();
for (i=0; i<1000; i++) {
    sprayContainer[i] = fullblock +
                        shellcode;
}
```

Let’s step back for a second

- We now know a number of attack techniques and defense mechanisms
- They are all focused on *control-data* attacks (and defenses)
  - Data that is eventually loaded in the program counter of the CPU; that is, that directly affects the control-flow of the program
- For an attacker, the goal is to take control of some control data
- For a defender, the goal is to protect control data from tampering

Control-data vs. non-control-data

- Can an attacker perform a meaningful attack by only controlling non-control-data (pure data)
- Attacker found a vulnerability, e.g., a stack-based buffer overflow
- There are restrictions on what he can do:
  - Leave alone saved RET
  - Only modify pure data
- What do we mean by meaningful?
  - Gaining the privileges of the vulnerable application

Non-Control-Data Attacks Are Realistic

- S. Chen et al., Non-Control-Data Attacks Are Realistic Threats, USENIX Security 2005 http://research.microsoft.com/pubs/73104/usenix05data_attack.pdf
- Many real-world applications can be exploited with non-control-data attacks and
- The severity of these attacks is equivalent to that of control-data attacks

Critical non-control data

- Configuration data
  - Path of files used by the program
- User identity data
  - User ID, group ID, corresponding access rights
  - Often cached in memory after program startup
  - Later access decision based on cached values
- User input
  - User input is validated and later used in security critical operation
- Decision making
  - Complex access control procedures may require multiple checks
  - Result of individual check may be stored as boolean variable

See the pattern: time-of-check vs. time-of-use (TOCTTOU)

GHTTPD stack overflow

- Web server
- Stack-based buffer overflow in log function
- Attacker goal
  - Execute /bin/sh via the CGI mechanism
  - Without the initial check, it would be trivial: /cgi-bin/../../../../bin/sh

```
serveconnection(int s) {
 char *ptr; // ptr to URL

 if (strstr(ptr, “/..”))
    reject_request(…);


 log(…);


 if (strstr(ptr, “cgi-bin”))
    handle_cgi_request(…);
}
```

GHTTPD stack overflow

```
log:
push %ebp
mov %esp, %ebp
push %edi
push %esi ; stores ptr
push %ebx
... stack buffer overflow
   code
pop %ebx
pop %esi
pop %edi
pop %ebp
ret
```

- Send input that passes check on /..
  - GET AAAA...wxyz\r\n /cgi-bin/../../../../bin/sh
- With the overflow, overwrite the value of %esi stored on the stack
  - New value: wxyz, the address of /cgi-bin/../../../../bin/sh
- When log returns, ptr points to /cgi-bin/../../../../bin/sh
- Handle_cgi_request will invoke /bin/sh

Take away points

- Defenses that solve the root cause of the vulnerability
  - No always practical
- Mitigation techniques for stack-based buffer overflows
  - Useful to make attacks less likely to succeed
- Non executable stack
  - return-into-libc
- Stack protection
- Address space randomization
  - Heap spraying
- Don’t ignore non-control-data attacks

Next time

- Other vulnerabilities
  - Heap overflow