Memory corruption defenses

Lecture 12
Secure Programming

Rescheduling

- Next week: no lectures

- We will likely have 2 hours of lecture on Monday, March 3, starting at 11:00am
  - I will make an announcement on Canvas when we have the final confirmation

Homework 3

- Homework 3 is out: http://www.cs.bham.ac.uk/~covam/teaching/2013/secprog/hw3.html

STACK PROTECTION

Linux implementation

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

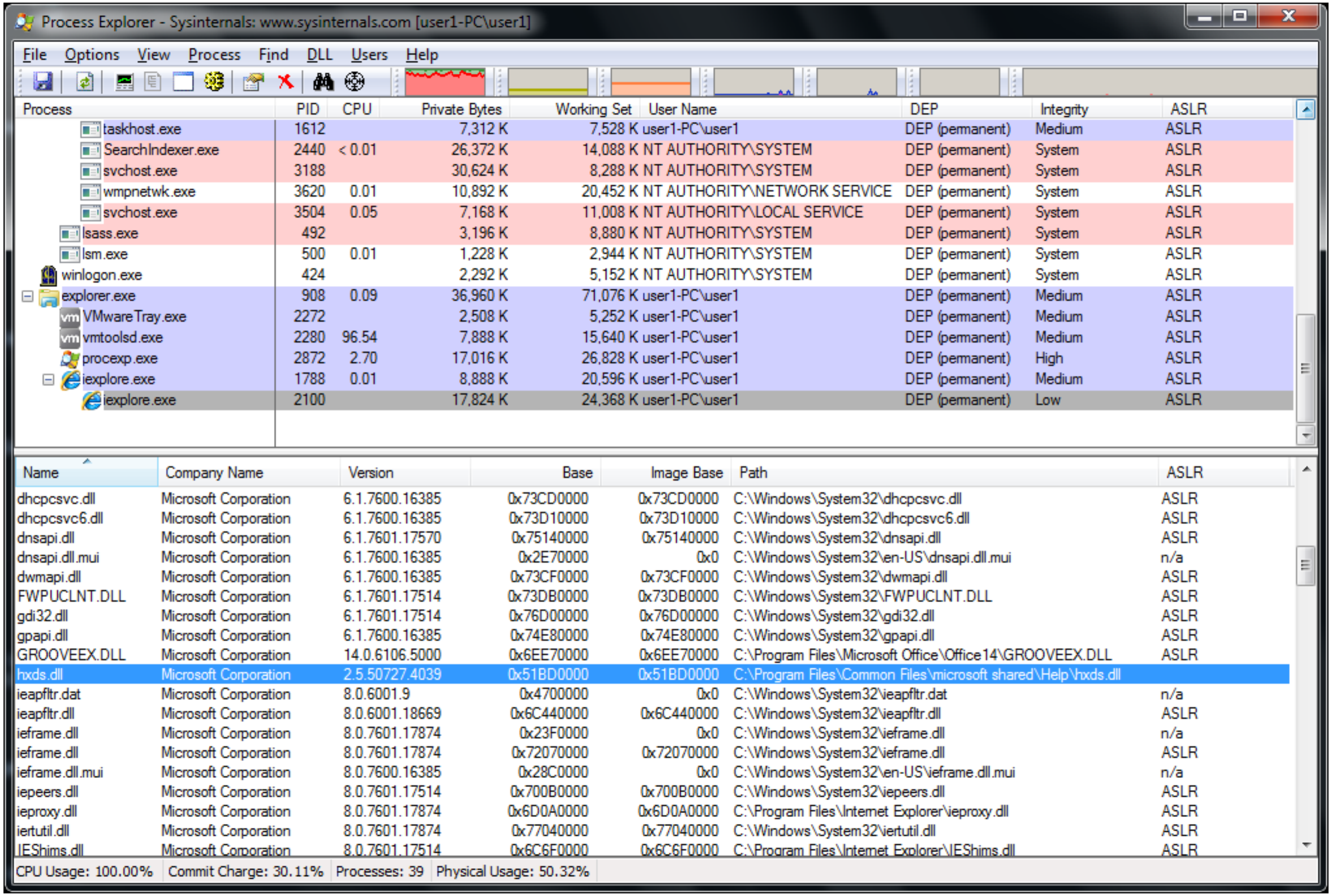
- Are all libraries on the system compiled to support ASRL?
- If not, can you (as an attacker) cause the non-ASLR library to be loaded in the address space of the vulnerable process?
- Defense is as strong as the weakest element of the defense system

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

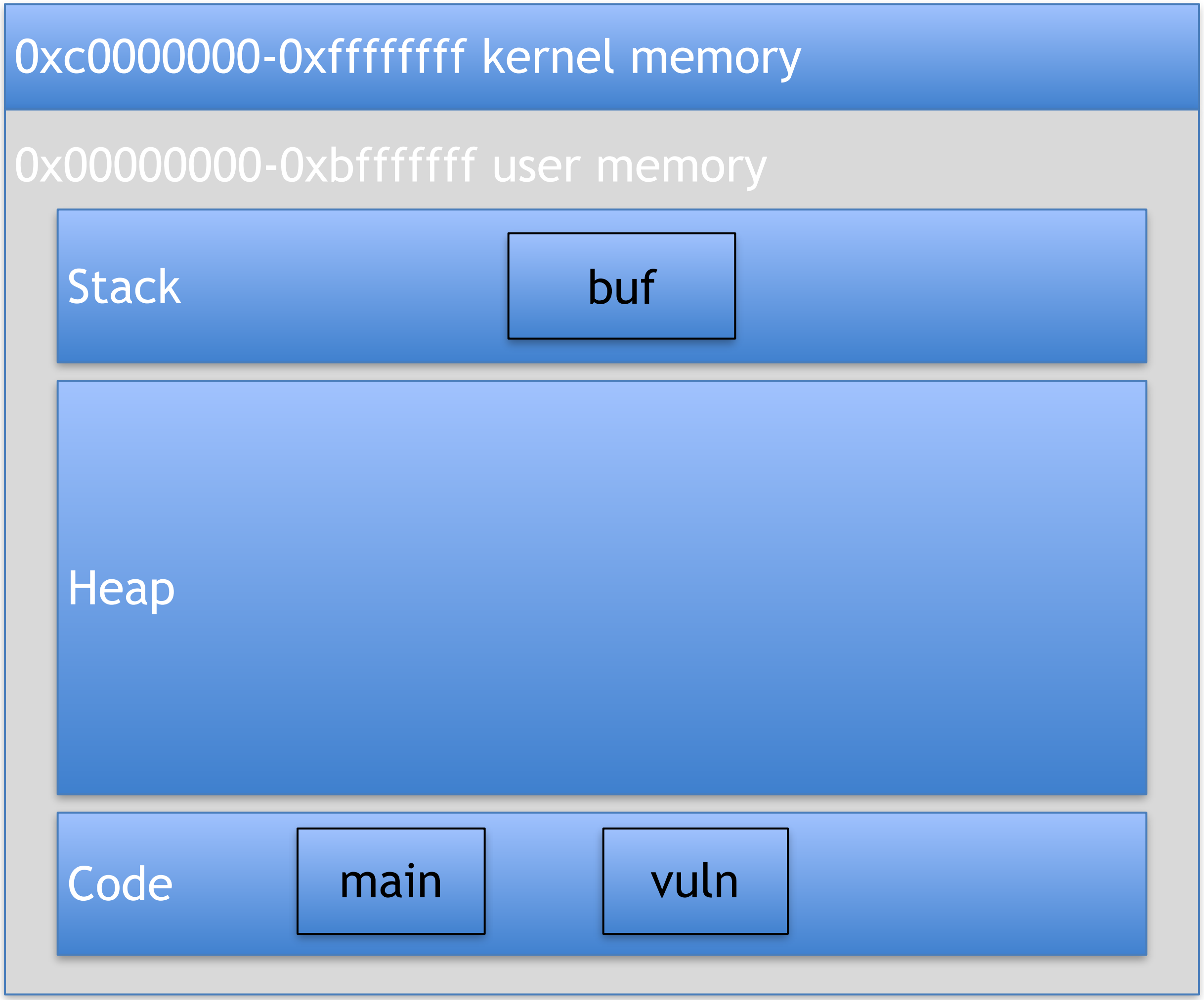


http://www.greyhathacker.net/?p=585

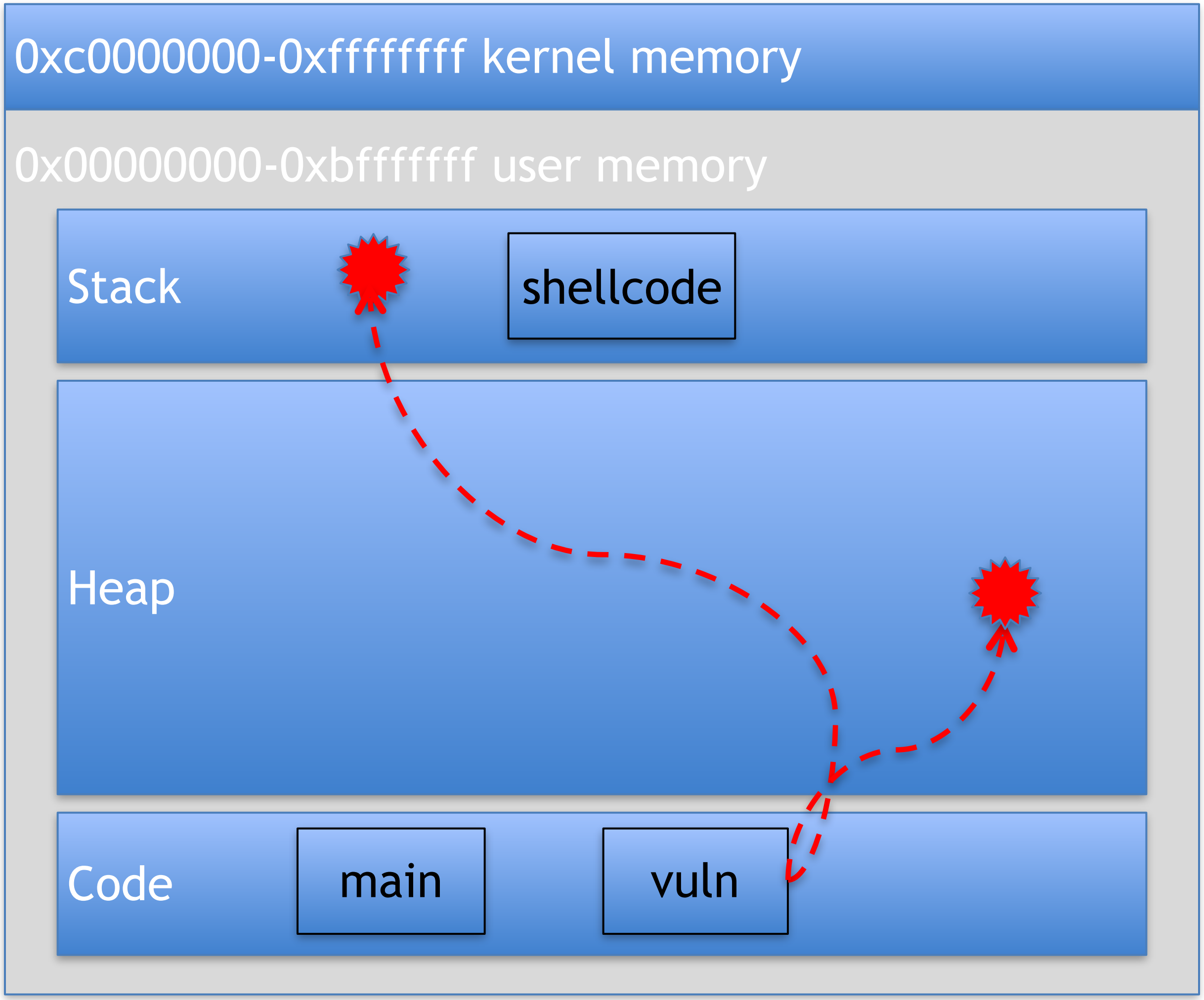
Bypassing ASLR (2)

- 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

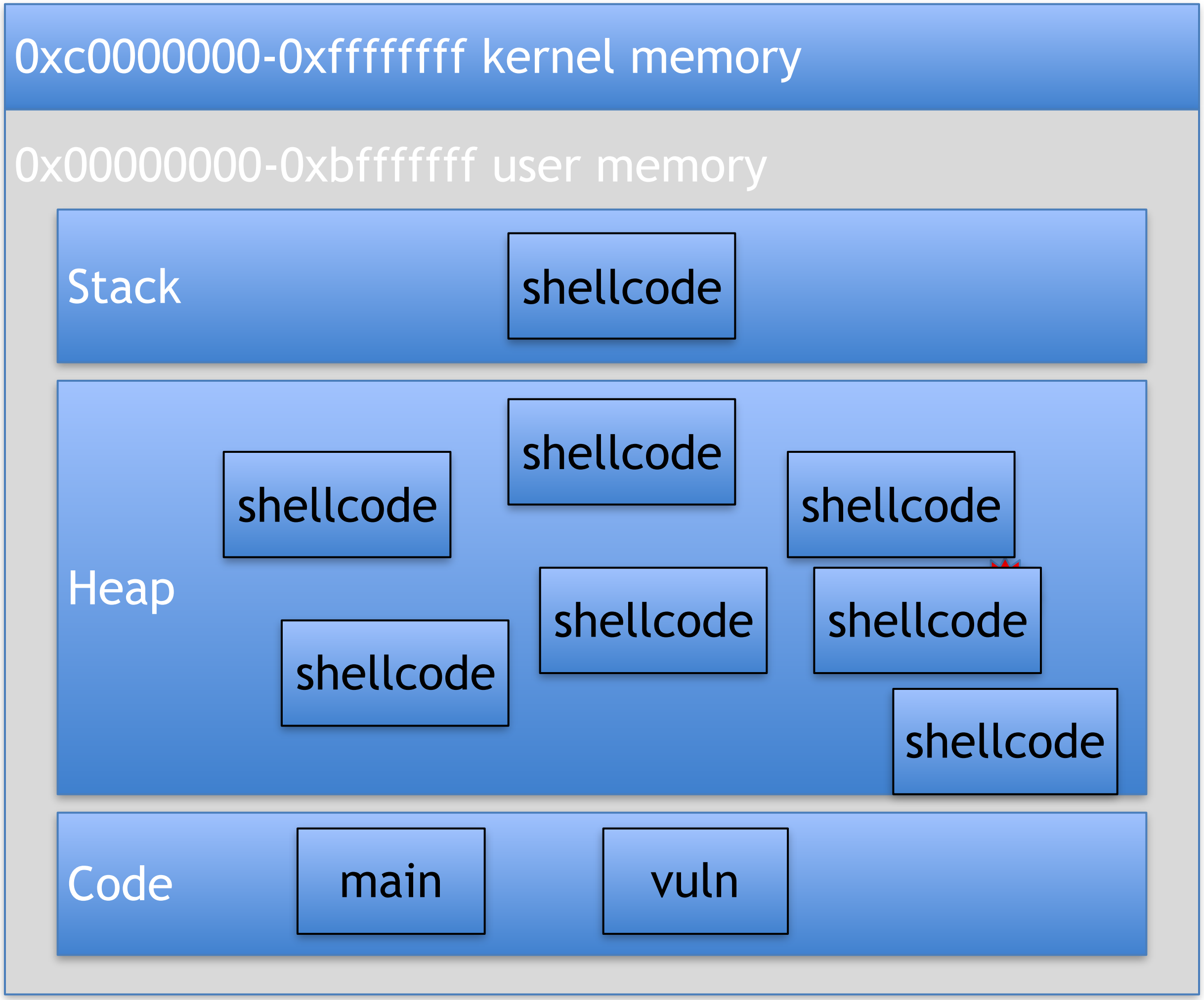
Heap spraying



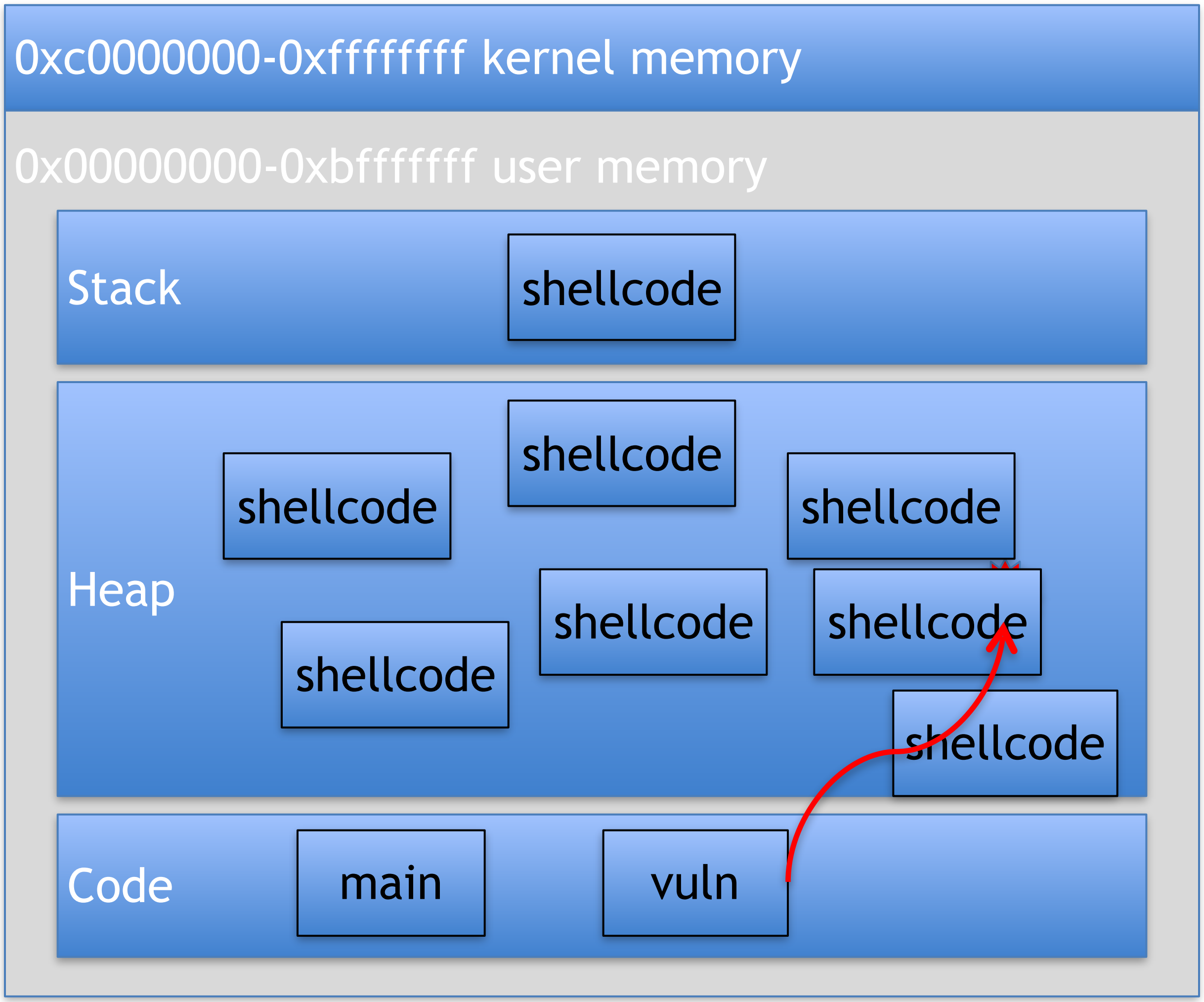
Heap spraying



Heap spraying



Heap spraying



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
- 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