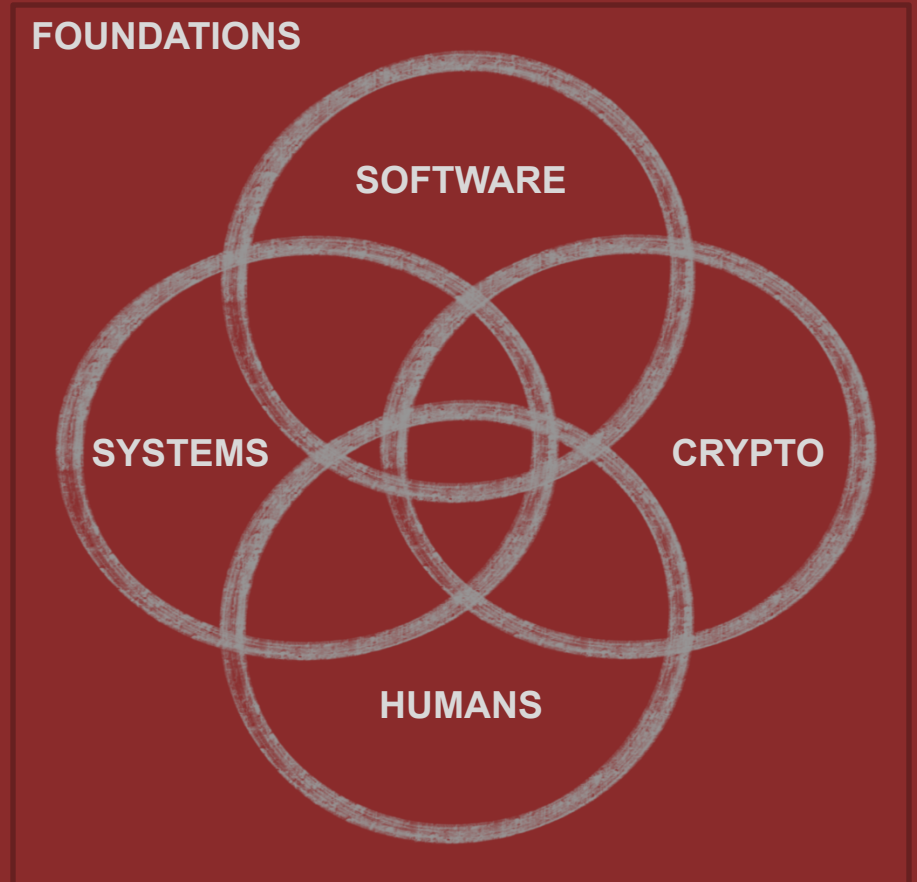


Διάλεξη #6 - Mitigations



Huge thank you to [David Brumley](#) from Carnegie Mellon University for the guidance and content input while developing this class

Την Προηγούμενη Φορά

1. CVEs
2. Format String Attacks and review

Ανακοινώσεις / Διευκρινίσεις


- Εργασία #1 θα βγει μέχρι το τέλος της εβδομάδας (ελπίζω)

Ερωτήσεις pending:

- Γιατί το πρόγραμμα "σκάει" ακόμα και όταν δεν κάνουμε overwrite το return address;
- Πρέπει να έχουμε shell access για να τρέξουμε κάποιο exploit;

Σήμερα

- Adversary and Classifications
- Mitigations



**Two Concepts
(Only a few I need
you to memorize)**

Defining The Adversary (1/2)

- Adversary = < Goals, Capabilities >
- Goal: What constitutes success?
 - May involve subgoals
 - Example goal: Gain control of X's data
 - Sub-goal: Reconnaissance: search online for info about X
 - Sub-goal: Access: Guess X's ssh password on Linux lab
 - » Sub-goal: Lateral movement: Use ssh account to move to other services / linked accounts
- Capabilities: What resources can the adversary use?
 - 1 computer or millions?
 - Physical or remote access?
 - Access to source code?

Why don't we include
adversary's strategy?

Defining The Adversary (1/2)


- Adversary = < Goals, Capabilities >



Why don't we include adversary's strategy?

Security Mechanism Classification for a property (2/2)

- 1. Prevention.** Prevent issues from happening. Any precautionary measures.
- 2. Detection.** Assuming an incident took place, detect them as early and as accurately as possible.
- 3. Resilience.** Assuming one or multiple incidents took place, ensure the overall system security degrades gracefully and does not collapse.
- 4. Deterrence.** Measures to ensure penalties for actors responsible for security incidents. Policy-based.



**Control Flow
Hijack Defenses /
Mitigations**

Defenses

1. Canaries
2. DEP (Data Execution Prevention) / NX (No Execute)
3. ASLR (Address Space Layout Randomization)

Defenses we will see today focus on **preventing** control hijacks (Prevention)

Γιατί το πρόγραμμα "σκάει" ακόμα και όταν δεν κάνουμε overwrite το return address;

```
int is_good() {  
    char * magic = "8675309";  
    char buf[32];  
    fread(buf, 128, 1, stdin); // BOFs are cool  
    if (strncmp(magic, buf, strlen(magic)) == 0) {  
        return 1;  
    }  
    return 0;  
}
```

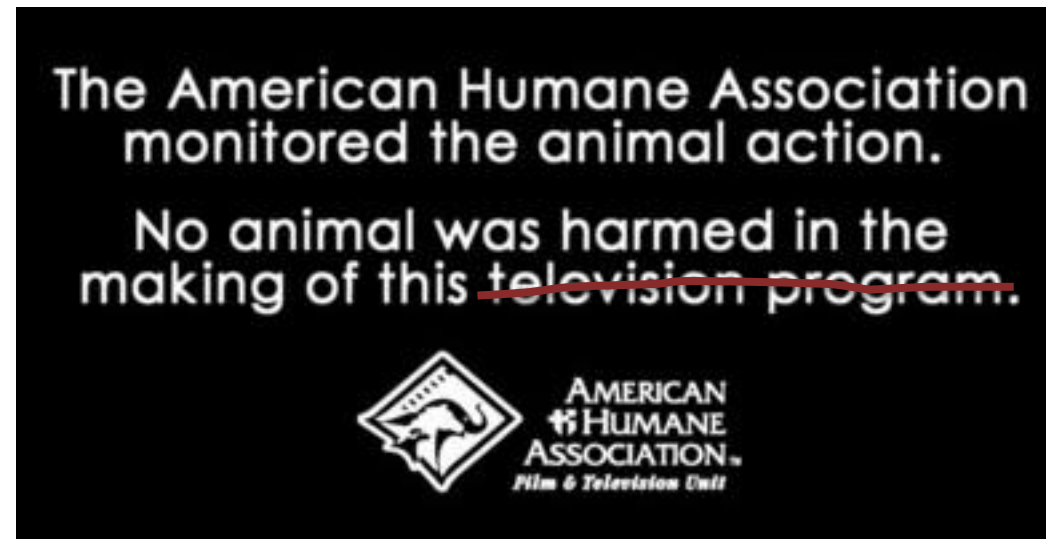
Μπορούμε να ελέγξουμε τον IP σε ένα πρόγραμμα σαν και αυτό; Γιατί ναι/όχι;

Canary / Stack Cookies



What Is a “Canary”?

Wikipedia: “the historic practice of using canaries in coal mines, since they would be affected by toxic gases earlier than the miners, thus providing a biological warning system.”



lecture

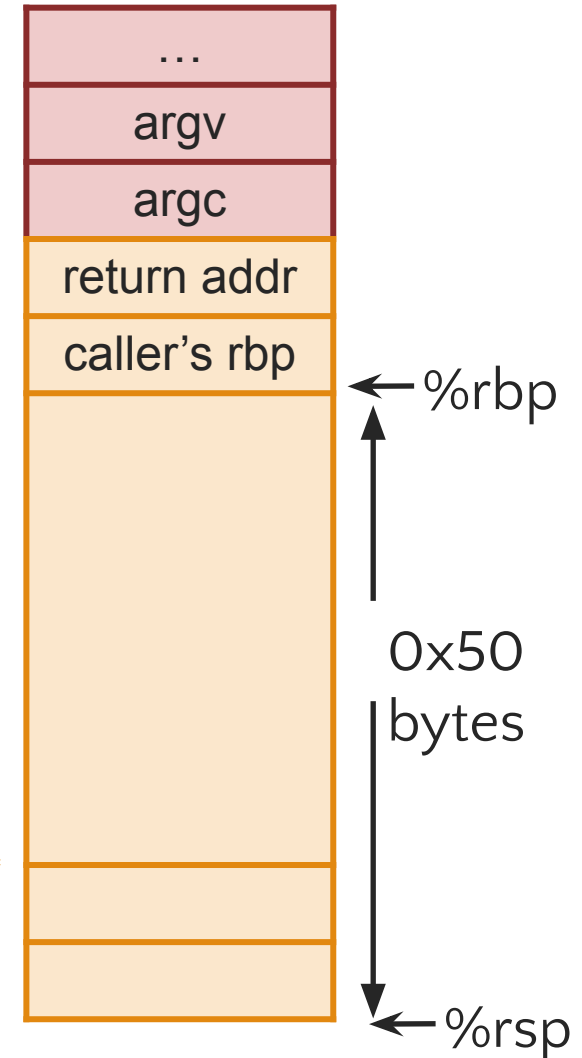
Reminder; Buffer Overflow

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
```

Dump of assembly code for function main:

```
4004fd: push    %rbp
4004fe: mov     %rsp,%rbp
400501: sub     $0x50,%rsp
400505: mov     %rdi,-0x48(%rbp)
400508: mov     %rsi,-0x50(%rbp)
40050c: lea    -0x40(%rbp),%rax
400510: mov     %rax,%rdi
400518: callq  400400 <gets@plt>
```

Reg	Value
rax	buf
rdi	buf



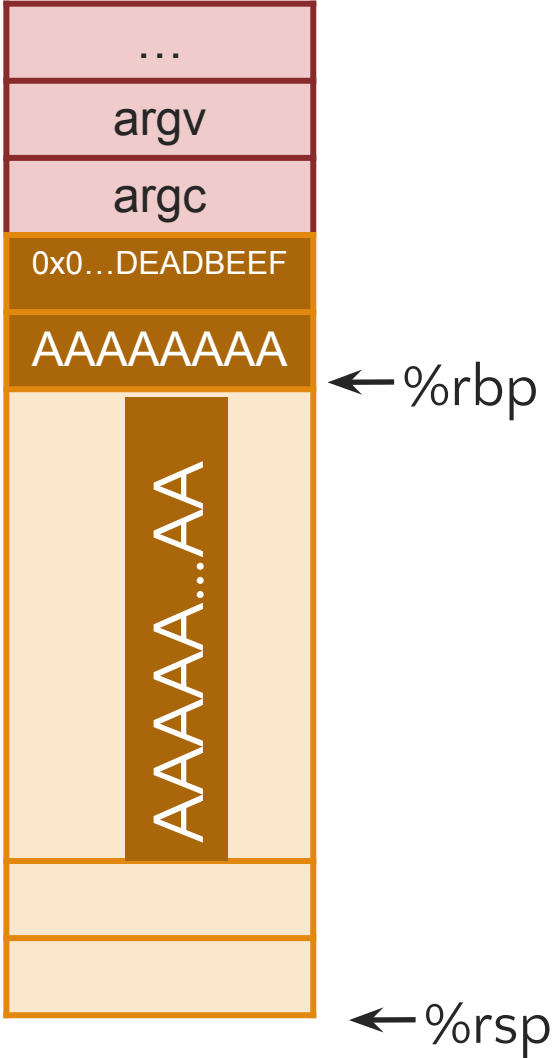
Input

“A”x72 + “\xEF\xBE\xAD\xDE\x00\x00\x00\x00”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
4004fd: push    %rbp
4004fe: mov     %rsp,%rbp
400501: sub     $0x50,%rsp
400505: mov     %rdi,-0x48(%rbp)
400508: mov     %rsi,-0x50(%rbp)
40050c: lea    -0x40(%rbp),%rax
400510: mov     %rax,%rdi
400518: callq  400400 <gets@plt>
40051d: leaveq
40051e: retq
```

*corrupted
overwritten
overwritten*

Reg	Value
rax	buf
rdi	buf



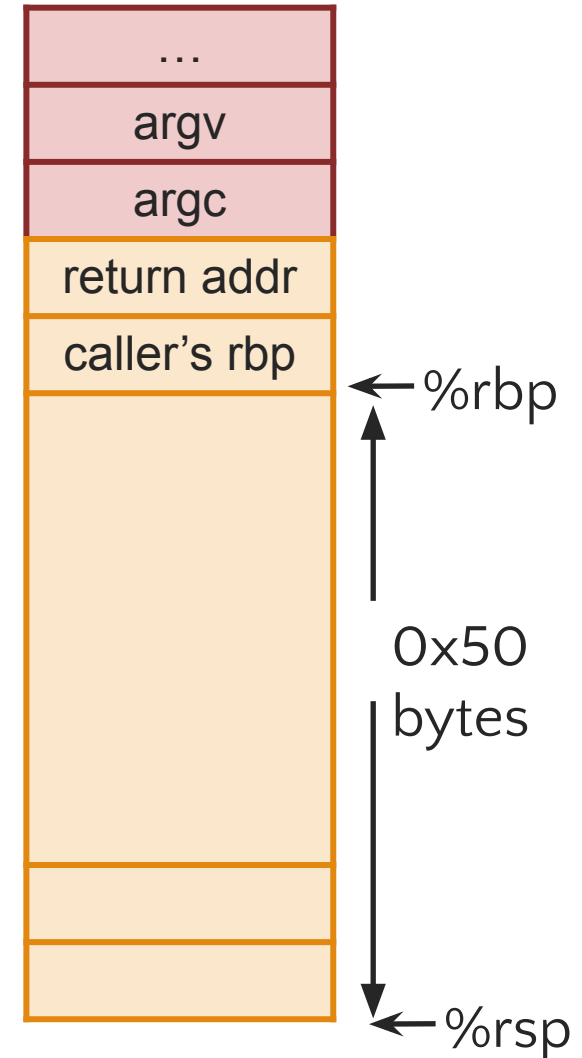
StackGuard

[Cowen et al. 1998]

Idea:

- prologue introduces a *canary word* between return addr and locals
- epilogue checks canary before function returns

Wrong canary => Overflow



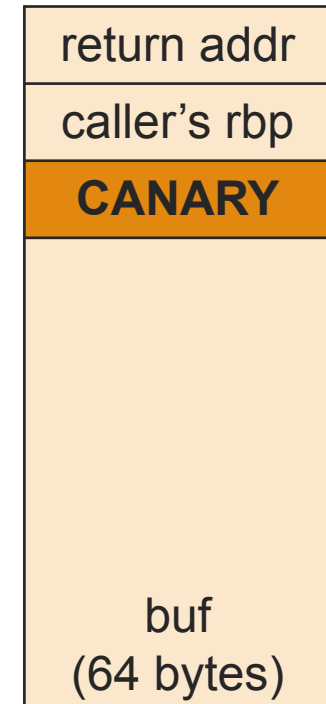
gcc Stack-Smashing Protector (ProPolice)

Dump of assembler code for function main:

```
4005a0: sub    $0x58,%rsp
4005a4: mov    %fs:0x28,%rax
4005ad: mov    %rax,0x48(%rsp)
4005b2: xor    %eax,%eax
4005b4: mov    %rsp,%rdi
4005b7: callq 4004a0 <gets@plt>
4005bc: mov    0x48(%rsp),%rdx
4005c1: xor    %fs:0x28,%rdx
4005ca: je     4005d1 <main+0x31>
4005cc: callq 400470 <__stack_chk_fail@plt>
4005d1: add    $0x58,%rsp
4005d5: retq
```

Compiled with `gcc -fstack-protector`

(you can also use `-fstack-protector-all` or `-fstack-protector-strong`)



Canary Should Be **HARD** to Forge



- Terminator canary
 - 4 bytes: 0,CR,LF,-1 (low->high)
 - terminate strcpy(), gets(), ...
- Random canary
 - 4 random bytes chosen at load time
 - stored in a guarded page
 - need good randomness

Proposed Defense Scorecard

Aspect	Defense
Performance	<ul style="list-style-type: none">• Smaller impact is better
Deployment	<ul style="list-style-type: none">• Can everyone easily use it?
Compatibility	<ul style="list-style-type: none">• Doesn't break libraries
Safety Guarantee	<ul style="list-style-type: none">• Completely secure vs. easy to bypass

Canary Scorecard

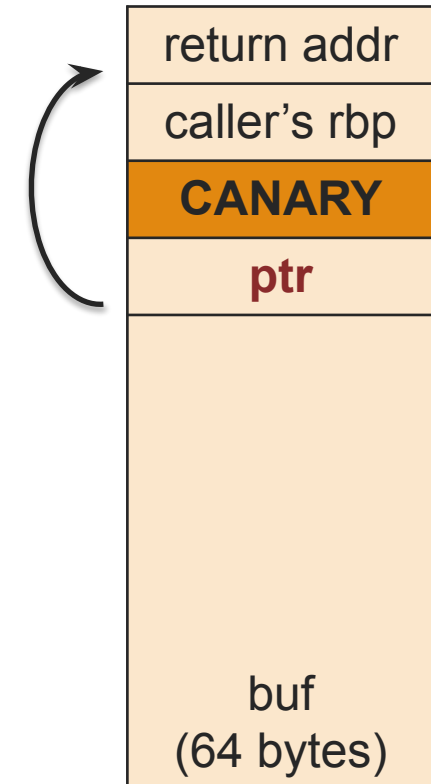
Aspect	Canary
Performance	<ul style="list-style-type: none">• several instructions per function• time: a few percent on average• size: can optimize away in safe functions (but see MS08-067 *)
Deployment	<ul style="list-style-type: none">• recompile suffices; no code change
Compatibility	<ul style="list-style-type: none">• perfect—invisible to outside
Safety Guarantee	<ul style="list-style-type: none">• <i>not really...</i>

[Shadow stack and canaries performance](#)

Bypass: Data Pointer Subterfuge

Overwrite a data pointer *first*...

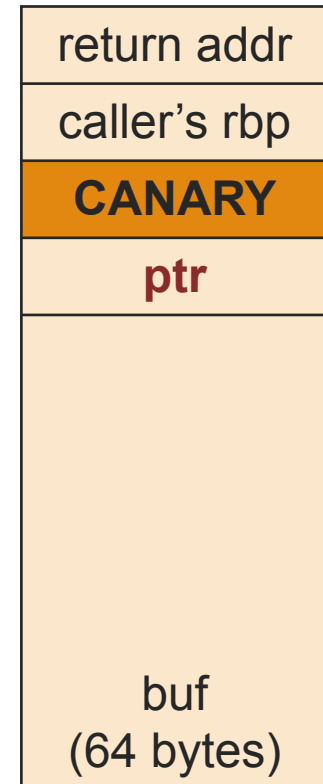
```
int *ptr;  
char buf[64];  
memcpy(buf, user1, large);  
*ptr = user2;
```



Bypass: Combine with a memory leak

Print out canary value first and use it in overwrite!

```
int *ptr;  
char buf[64];  
printf(user2);  
memcpy(buf, user1, large);
```



Canary Weakness

Check does *not* happen until epilogue...

- func ptr subterfuge
- C++ vtable hijack
- exception handler hijack
- ...

} PointGuard
} SafeSEH
} SEHOP

} ProPolice
puts arrays
above others
when possible

└──────────┘
struct is fixed;
& what about heap?

Quiz

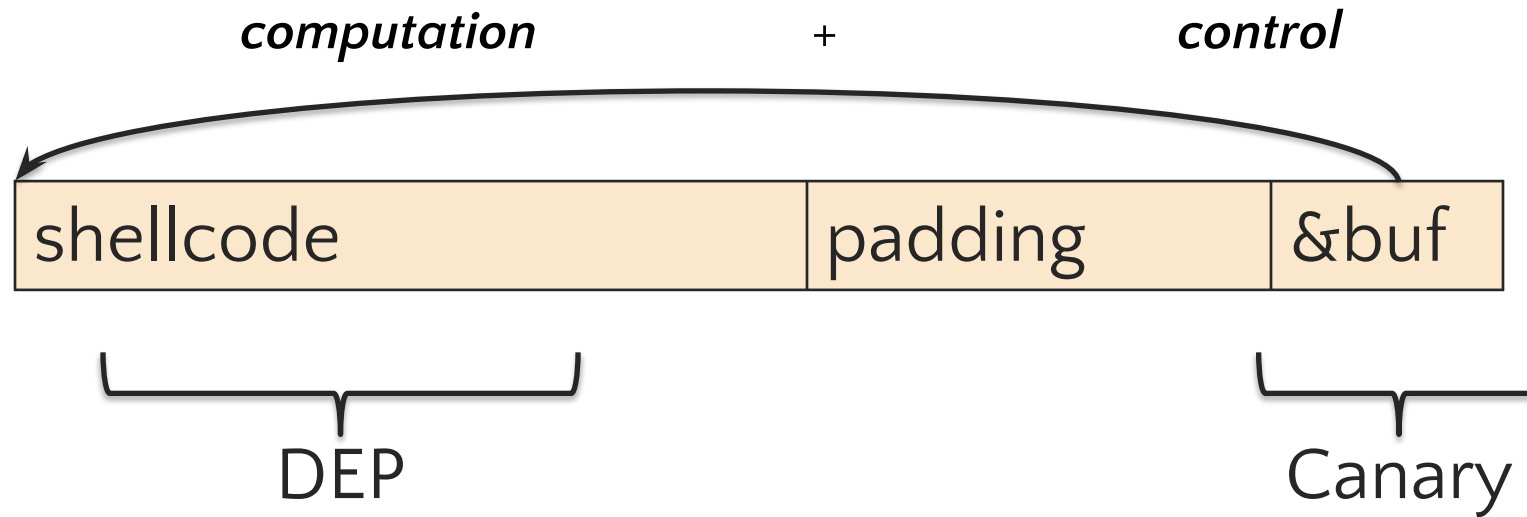
Which attack would be ***MOST*** effective at **hijacking control on a canary-protected machine?**

- A. Using a **single** `memcpy`-based buffer overflow of a local variable
- B. Using a format-string vulnerability **and** the “%n” specifier
- C. Using a format-string vulnerability **and** a targeted address specifier (e.g., “%9\$sBBBB\x47\xf7\xff\xff”)
- D. Using a format-string overflow of a local variable (e.g., “%80u\x3c\xd3\xff\xff”)

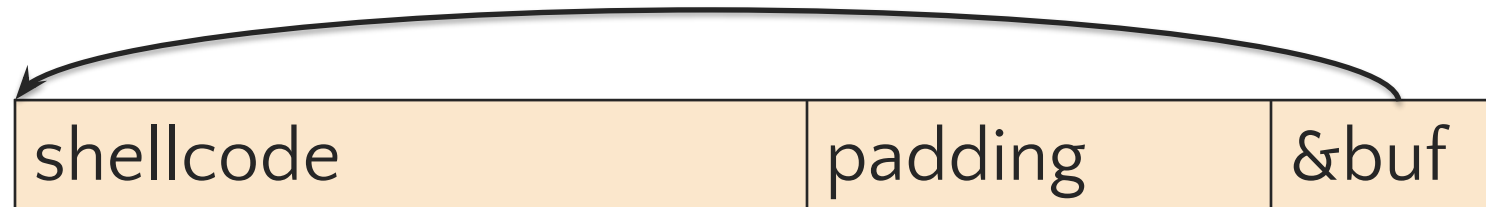
Data Execution Prevention (DEP) / No eXecute (NX)

Idea: maybe we shouldn't allow data to be executable

How to Defeat Exploits?



Data Execution Prevention



Mark stack as
non-executable
using NX bit

(still a Denial-of-Service attack!)

DEP prevents injected code on the
stack from executing

DEP Scorecard

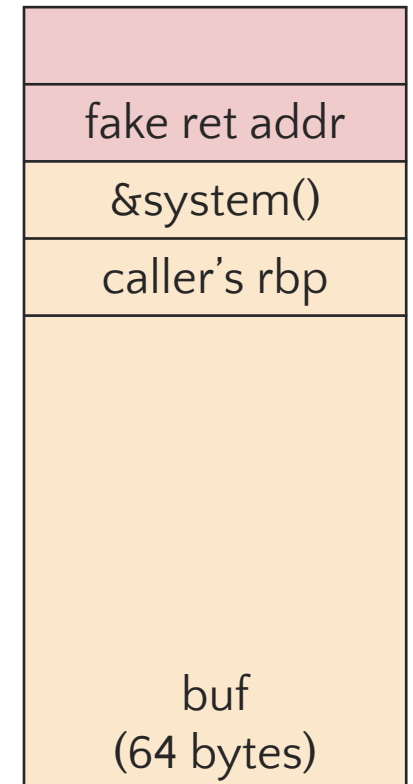
Aspect	Data Execution Prevention
Performance	<ul style="list-style-type: none">• with hardware support: no impact• otherwise: reported to be <1% in PaX
Deployment	<ul style="list-style-type: none">• kernel support (common on all platforms)• modules opt-in (less frequent in Windows)
Compatibility	<ul style="list-style-type: none">• can break legitimate programs<ul style="list-style-type: none">- Just-In-Time compilers- unpackers
Safety Guarantee	<ul style="list-style-type: none">• code injected to NX pages never execute• <i>but code injection may not be necessary...</i>

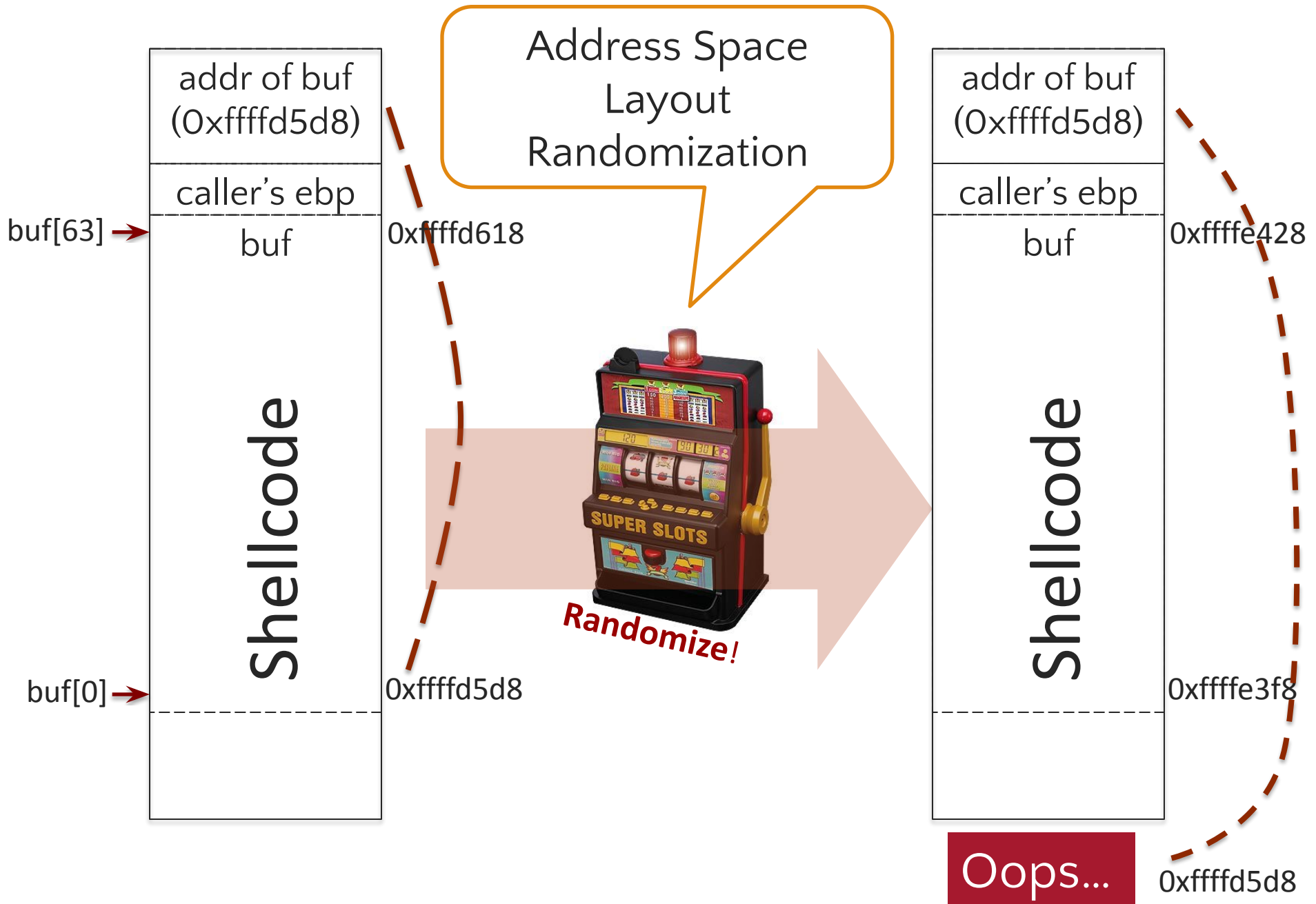
Return-to-libc Attack

Overwrite return address with the address of a libc function

- setup fake return address
- put arguments (e.g. “/bin/sh”) in correct registers / memory
- ret will “call” libc function

No injected code!





ASLR

Traditional exploits need precise addresses

- *stack-based overflows*: location of shell code
- *return-to-libc*: library addresses
- **Problem:** program's memory layout is fixed
 - stack, heap, libraries etc.
- **Solution:** randomize addresses of each region!

Running cat Twice

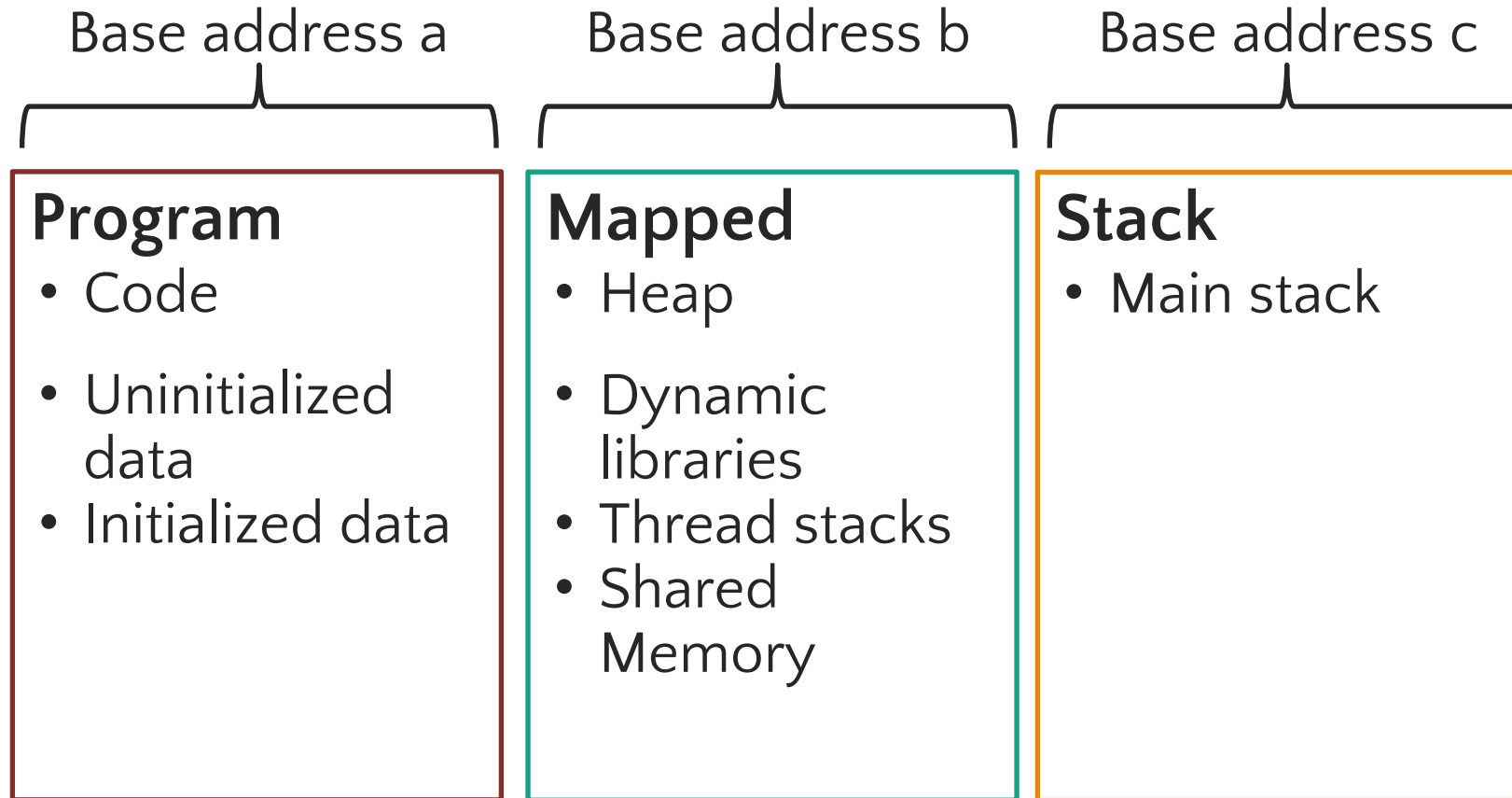
- Run 1

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'  
082ac000-082cd000 rw-p 082ac000 00:00 0 [heap]  
b7dfe000-b7f53000 r-xp 00000000 08:01 1750463 /lib/i686/cmov/libc-2.7.so  
b7f53000-b7f54000 r--p 00155000 08:01 1750463 /lib/i686/cmov/libc-2.7.so  
b7f54000-b7f56000 rw-p 00156000 08:01 1750463 /lib/i686/cmov/libc-2.7.so  
bf966000-bf97b000 rw-p bffeb000 00:00 0 [stack]
```

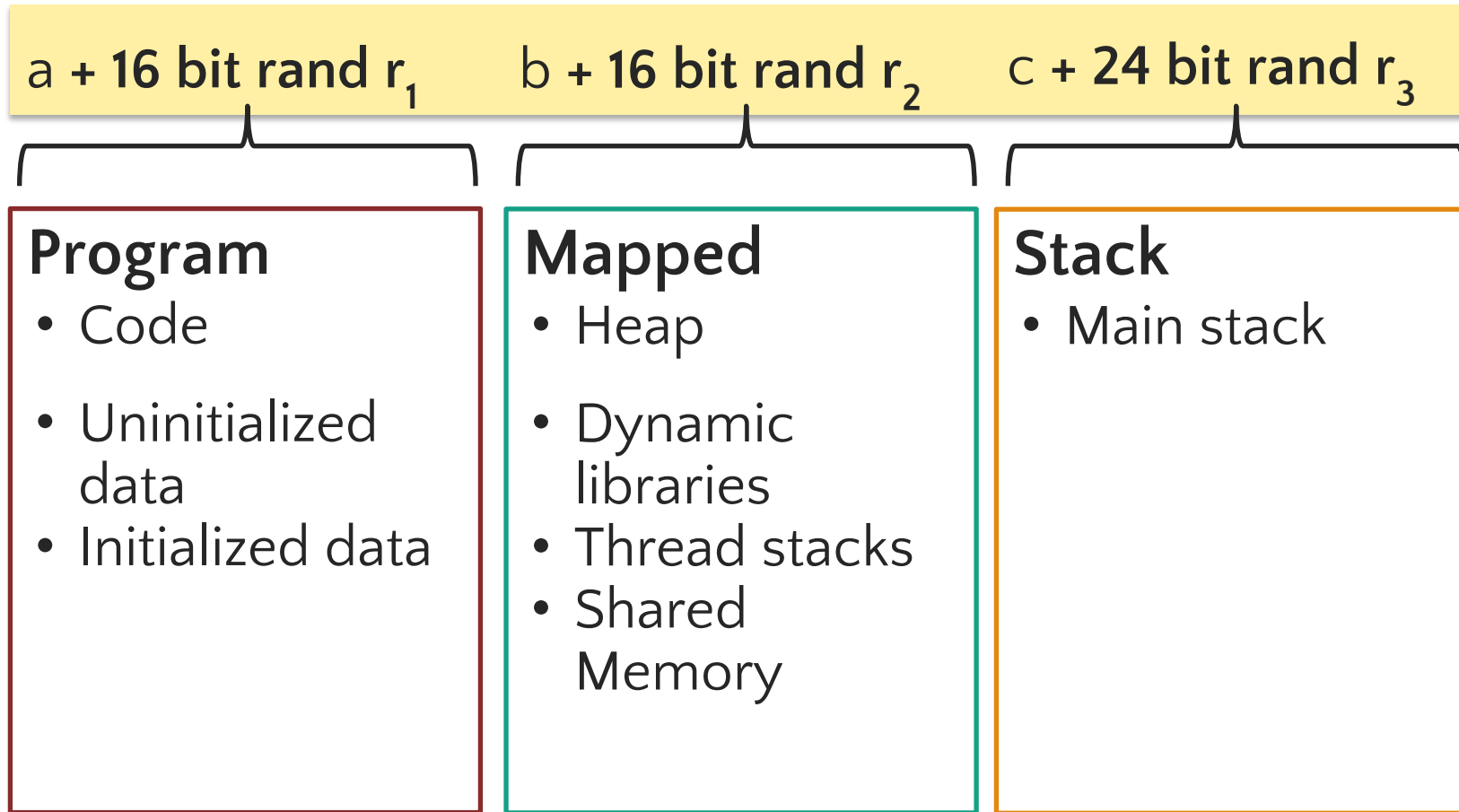
- Run 2

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'  
086e8000-08709000 rw-p 086e8000 00:00 0 [heap]  
b7d9a000-b7eef000 r-xp 00000000 08:01 1750463 /lib/i686/cmov/libc-2.7.so  
b7eef000-b7ef0000 r--p 00155000 08:01 1750463 /lib/i686/cmov/libc-2.7.so  
b7ef0000-b7ef2000 rw-p 00156000 08:01 1750463 /lib/i686/cmov/libc-2.7.so  
bf902000-bf917000 rw-p bffeb000 00:00 0 [stack]
```


Memory



ASLR Randomization



* \approx 16 bit random number of 32-bit system. More on 64-bit systems.

ASLR Scorecard

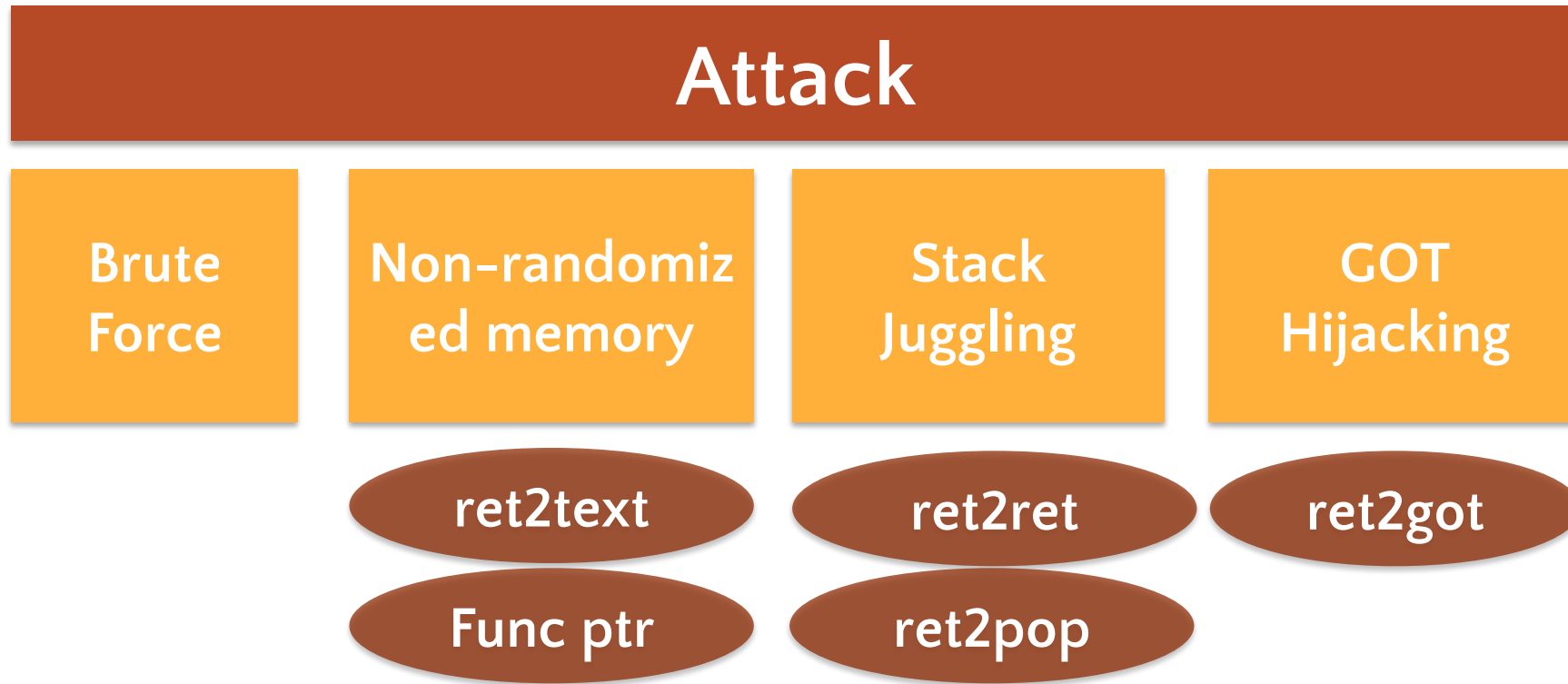
Aspect	Address Space Layout Randomization
Performance	<ul style="list-style-type: none">• excellent—randomize once at load time
Deployment	<ul style="list-style-type: none">• turn on kernel support (Windows: opt-in per module, but system override exists)• no recompilation necessary
Compatibility	<ul style="list-style-type: none">• transparent to safe apps (position independent)
Safety Guarantee	<ul style="list-style-type: none">• not good on x32, much better on x64• <i>code injection may not be necessary...</i>

Ubuntu - ASLR

- ASLR is **ON** by default [Ubuntu-Security]
 - `cat /proc/sys/kernel/randomize_va_space`
 - In older systems: **1** (*stack/mmap* ASLR)
 - In later releases: **2** (*stack/mmap/brk* ASLR)
 - `stack/mmap/brk/exec` ASLR: available since 2008 - still systems around without it
 - Position Independent Executable (PIE) with “`-fPIE -pie`”

Remember: you probably want this enabled

How to Attack ASLR?



More to Come Later

return-Oriented
PROGRAMMING

Ευχαριστώ και καλή μέρα εύχομαι!

Keep hacking!