Lec 12: Memory Defense

CS492E: Introduction to Software Security

Sang Kil Cha

Defense #1: NX

NX (No eXecute)

a.k.a. Data Execution Prevention* (***DEP***)

Each memory page has different ***R***ead, ***W***rite, e***X***ecute permissions.

We can put the stack on a separate page with no-executable permission to mitigate stack-based exploits. (e.g., Linux PaX)

* DEP ***prevents*** data execution, but it does not prevent buffer overflows.

NX (No eXecute)

a.k.a. Data Execution Prevention (***DEP***)

AMD Athlon™ Processor Competitive Comparison

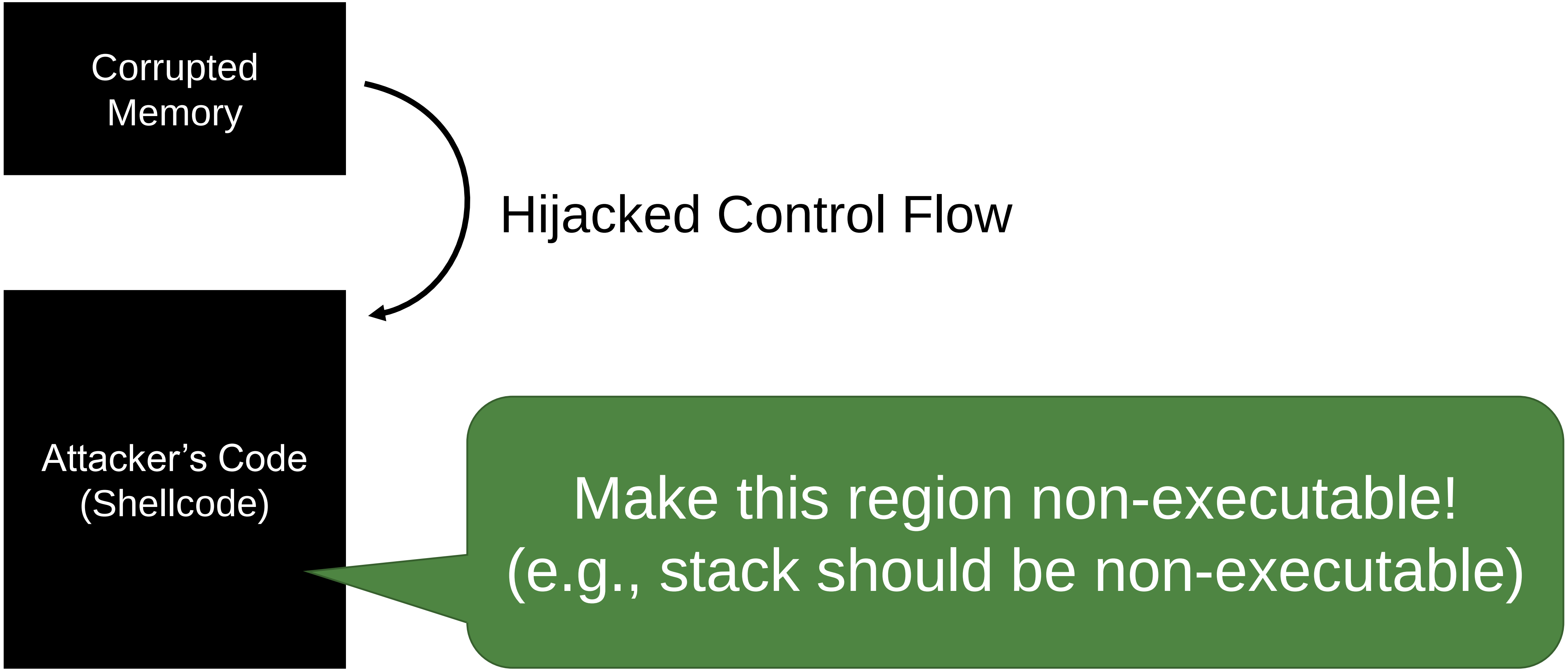
FEATURES	AMD ATHLON™ CPU	PENTIUM® 4
Architecture Introduction	2006	2000
Infrastructure	Socket AM2	Socket LGA775
Process Technology	90 nanometer, SOI 65 nanometer, SOI	90 nanometer
64-bit Instruction Set Support	Yes, AMD64 technology	Depends, EM64T on some Pentium® 4 series
Enhanced Virus Protection for Windows® XP SP2*	Yes	Depends

W^X (Write XOR eXecute) Policy

- Every page is either writable or executable, but not both!

- Even though we can put a shellcode (write) to a buffer, we cannot execute it if W^X is enabled.

Defeating Control Flow Hijack with DEP



execstack

- Tool to set, clear, or query NX stack flag of binaries

```
$ /usr/sbin/execstack -s <filename> ; clear NX flag
$ /usr/sbin/execstack -c <filename> ; set NX flag
$ /usr/sbin/execstack -q <filename> ; query NX flag
```

When NX is set, return-to-stack exploit will fail
(i.e., the program will crash)

New Attack

Bypassing DEP

- Return-to-stack exploit is disabled.

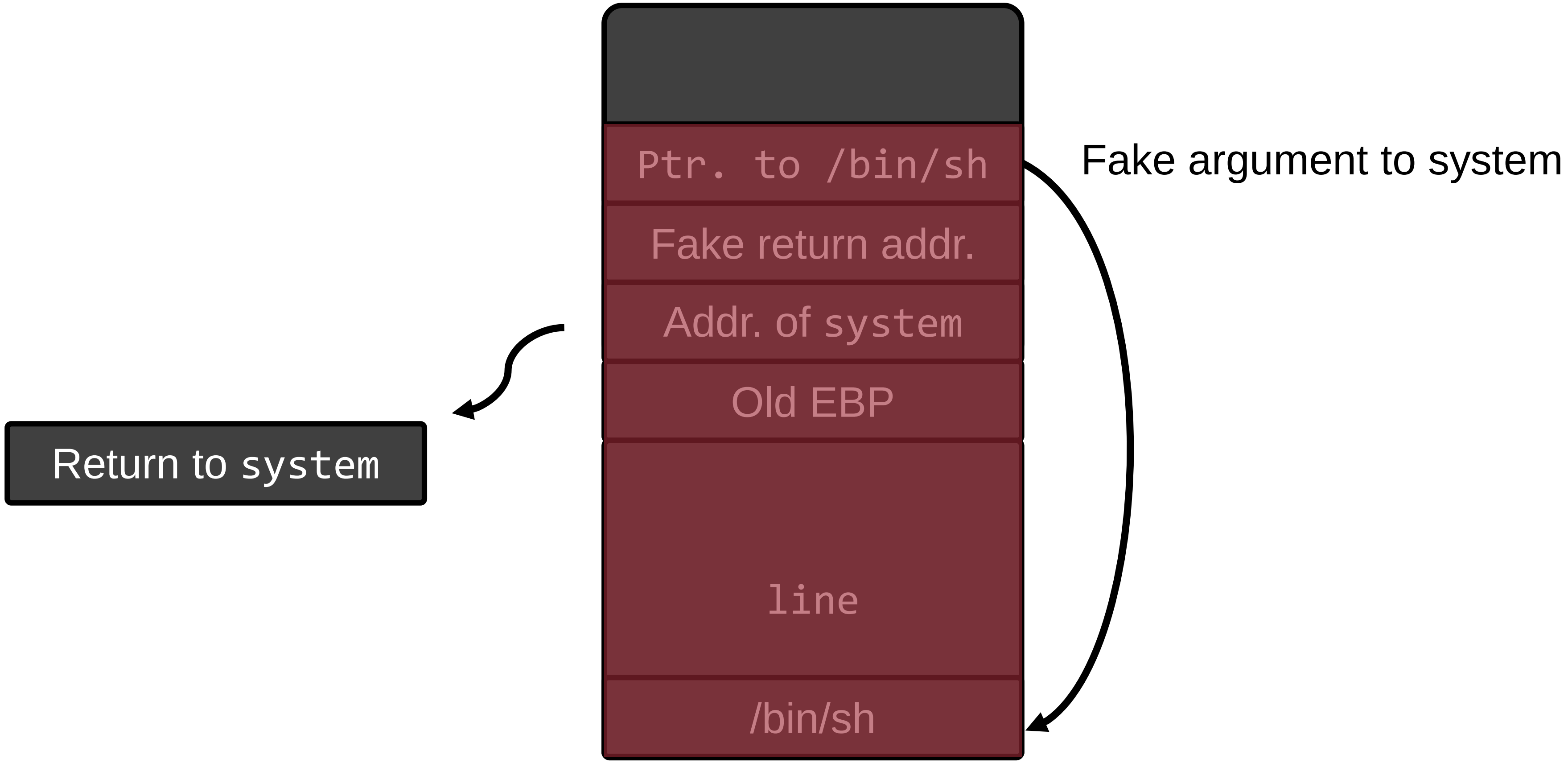
- But we can still jump to an arbitrary address of existing code.
  (= ***Code Reuse Attack***)

Code Reuse Attack #1: Return-to-Libc

- LIBC is a standard library that most programs commonly use
  - For example, `printf` is in LIBC

- Many useful functions in LIBC to return to
  - `exec` family (`execl`, `execlp`, `execle`, …)
  - `system`
  - `mprotect`
  - `mmap`

Return-to-Libc



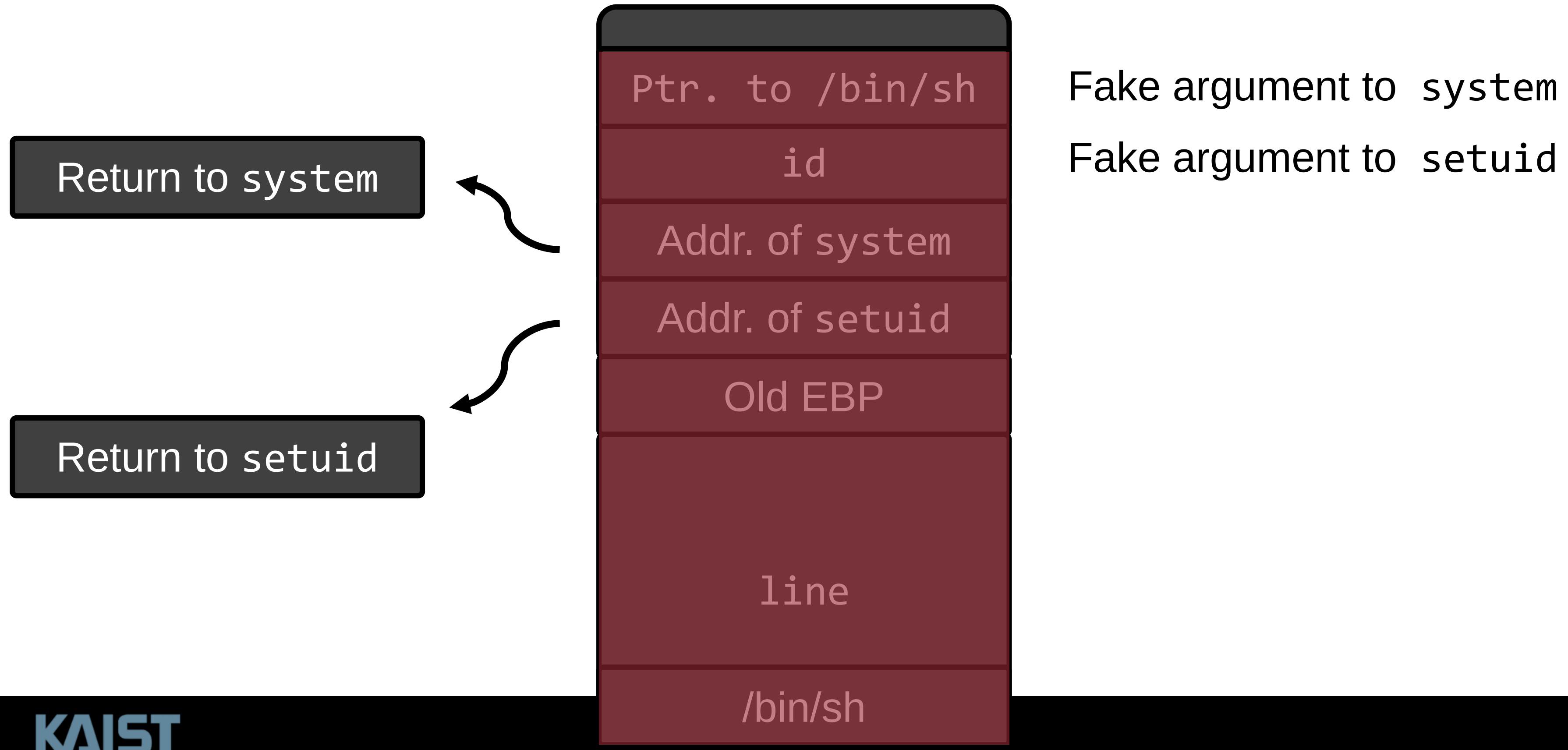
No injected shellcode!

Question

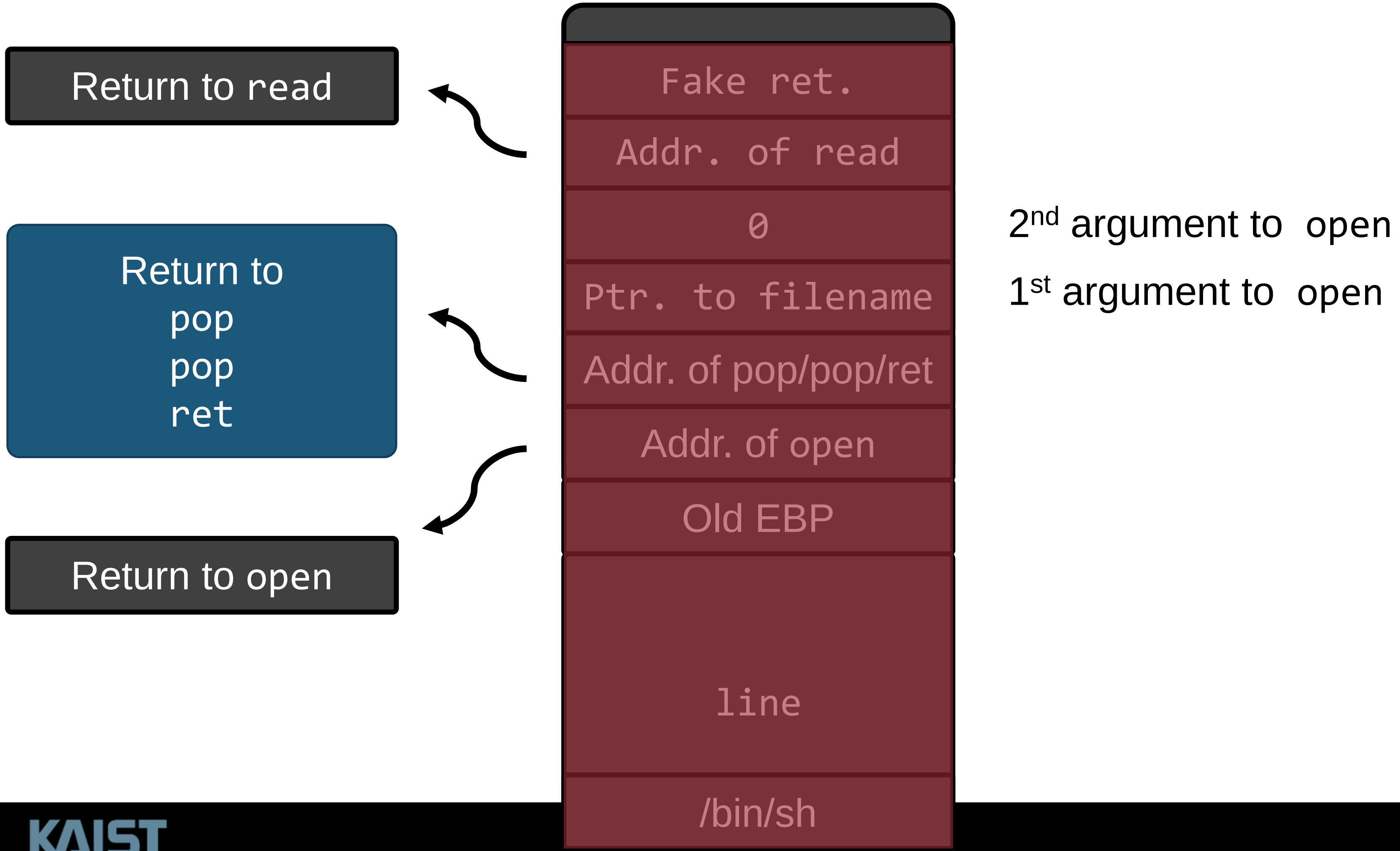
Can we call multiple LIBC functions? For example, we want to call `setuid()` first and then call `execve()`.

Return-to-Libc: Two Function Calls

What if the first function call to LIBC requires more than one parameter?



Chaining Multiple Function Calls (ESP lifting)



Return to

```
pop
pop
ret
```

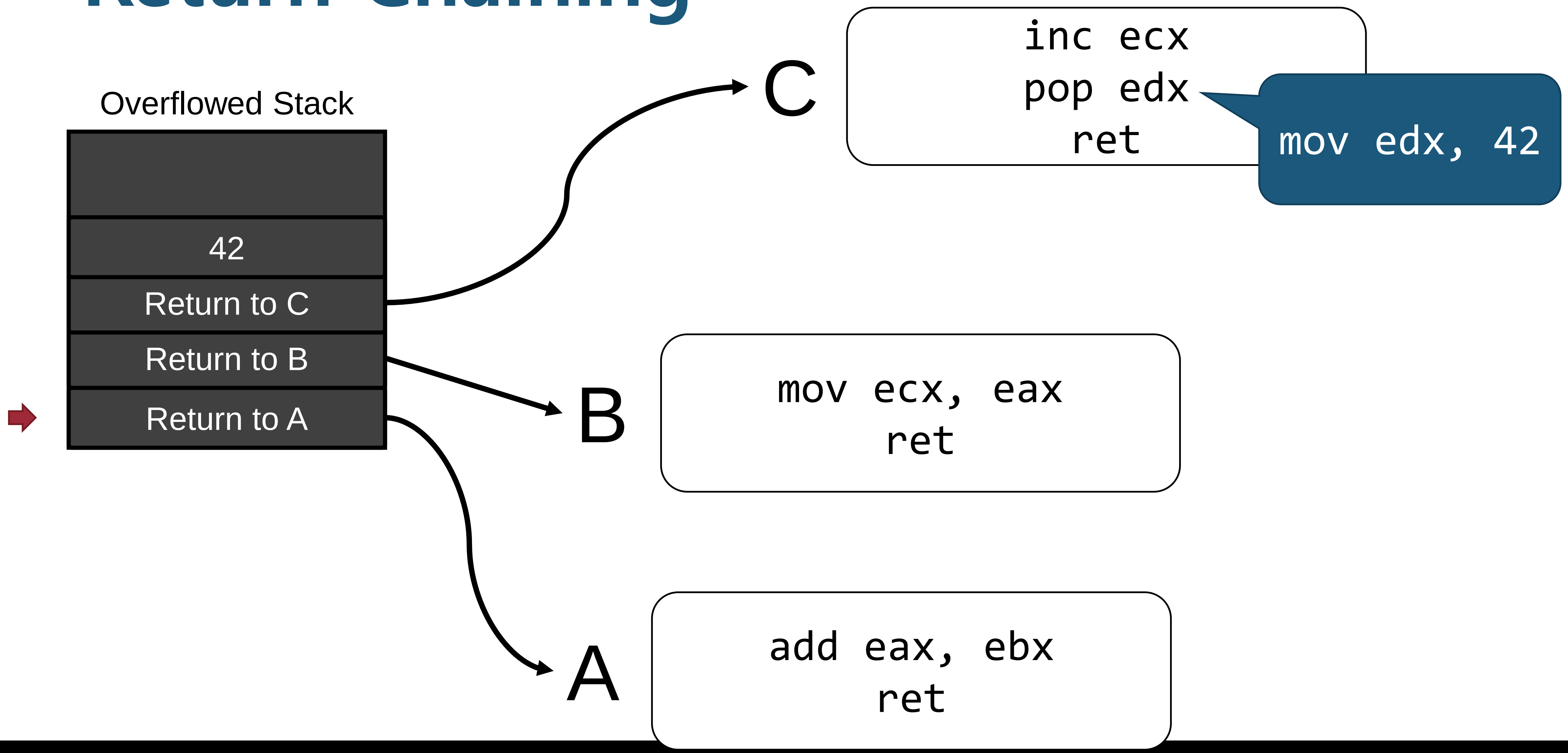
The idea of jumping into a code block that ends with "`ret`" instruction, becomes the primitive of ROP (Return-Oriented Programming)

Return-Oriented Programming (ROP)

Generalized version of Code Reuse Attack.

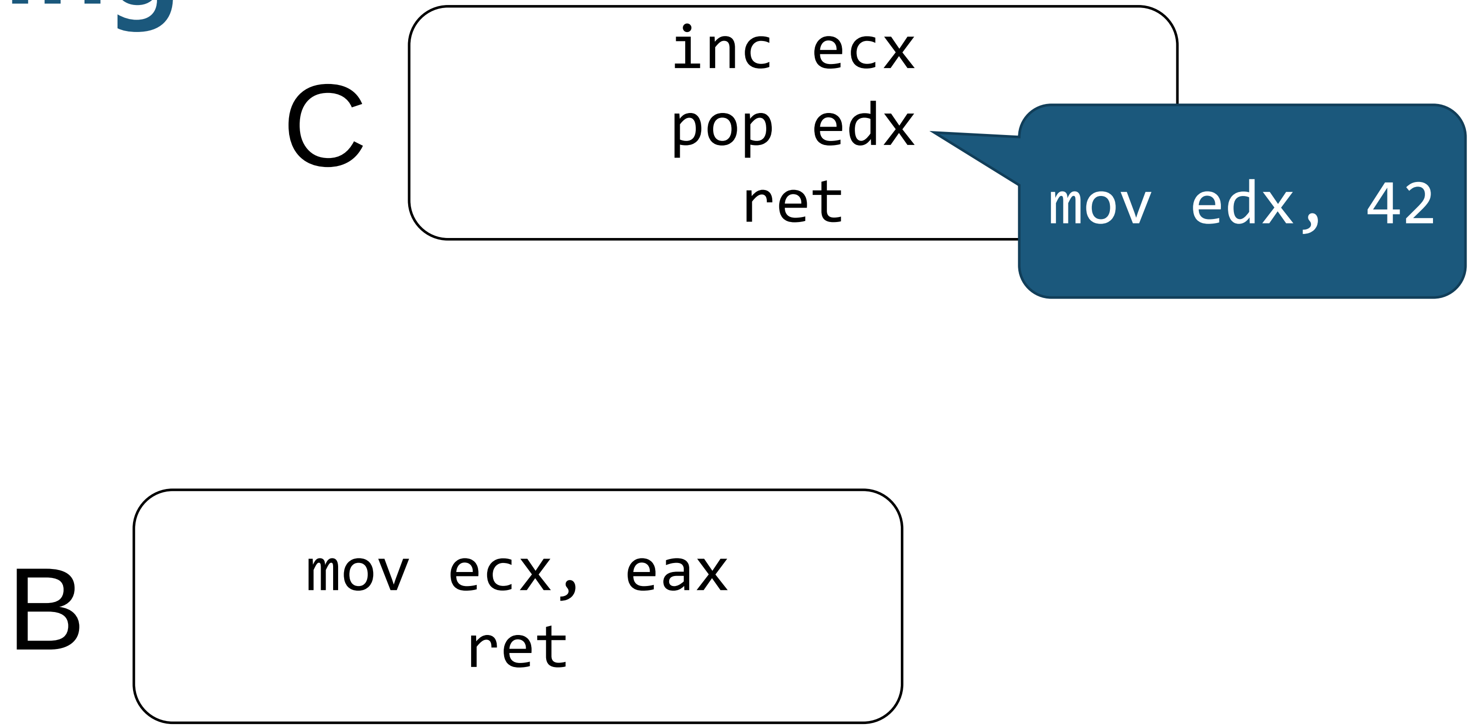
Formally introduced by Hovav in CCS 2007.
The Geometry of Innocent Flesh on the Bone: Return-to-libc without Function Calls (on the x86)

Return Chaining



Return Chaining

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Return chaining allows arbitrary computation!

ROP Practice

Our Goal:

Modify *ptr*, a function pointer, to be `0x42424242`

```
mov [ptr], 0x42424242
```

ROP Practice

```
mov [ptr], 0x42424242
```

`pop eax` `ret`	Get the *ptr*
`pop ebx` `ret`	Get 0x42424242
`mov [eax], ebx` `ret`	Modify the *ptr*

ROP Workflow

1. Disassemble binary
2. Identify useful instruction sequences (often called gadgets)
   - E.g., an instruction sequence that ends with `ret` is useful
   - E.g., an instruction sequence that ends with `jmp reg` can be useful (`pop eax; jmp eax`)
3. Assemble gadgets to perform some computation
   - E.g., spawning a shell

Disassembling x86

x86 instructions have variable lengths

```
08048aac <main>:
 8048aac:       8d 4c 24 04             lea     ecx,[esp+0x4]
 8048ab0:       83 e4 f0                and     esp,0xfffffff0
 8048ab3:       ff 71 fc                push    DWORD PTR [ecx-0x4]
 8048ab6:       55                      push    ebp
 8048ab7:       89 e5                   mov     ebp,esp
 8048ab9:       51                      push    ecx
 8048aba:       83 ec 14                sub     esp,0x14
 8048abd:       c7 45 f0 88 ad 0a 08    mov     DWORD PTR [ebp-0x10],0x80aad88
 8048ac4:       c7 45 f4 00 00 00 00    mov     DWORD PTR [ebp-0xc],0x0
 8048acb:       83 ec 04                sub     esp,0x4
 8048ace:       6a 00                   push    0x0
 8048ad0:       8d 45 f0                lea     eax,[ebp-0x10]
 8048ad3:       50                      push    eax
 8048ad4:       68 88 ad 0a 08          push    0x80aad88
 8048ad9:       e8 02 39 01 00          call    805c3e0 <__execve>
 ...
```

Disassembling x86

```
8d 4c 24 04 83 e4 f0 …
```

```
lea ecx,[esp+0x4]
and esp,0xfffffff0
```

What if we disassemble the code from the second byte (4c)?

Disassembling x86

```
8d 4c 24 04 83 e4 f0 …
```

```
dec esp
and al, 0x4
and esp,0xfffffff0
```

Totally different, but still *valid* instructions!

Disassemble from Any Addresses in Memory Pages

- This is perfectly legal
- We can find lots of ***unintended*** `ret` instructions

Unintended `ret` Instructions

Compiler intended instructions:

```
e8 05 ff ff ff          call    8048330
81 c3 59 12 00 00       add     ebx,0x1259
```

If we disassemble the above starting from the 2nd byte:

```
05 ff ff ff 81          add     eax,0x81ffffff
c3                      ret
```

Defense #2: ASLR

World without ASLR
(Address Space Layout Randomization)

Same address space over and over again.
(The only thing that matters was environment variable)

Printing out ESP

```
#include <stdio.h>

int main(void)
{
  int x = 42;
  return printf("%08x\n", &x);
}
```

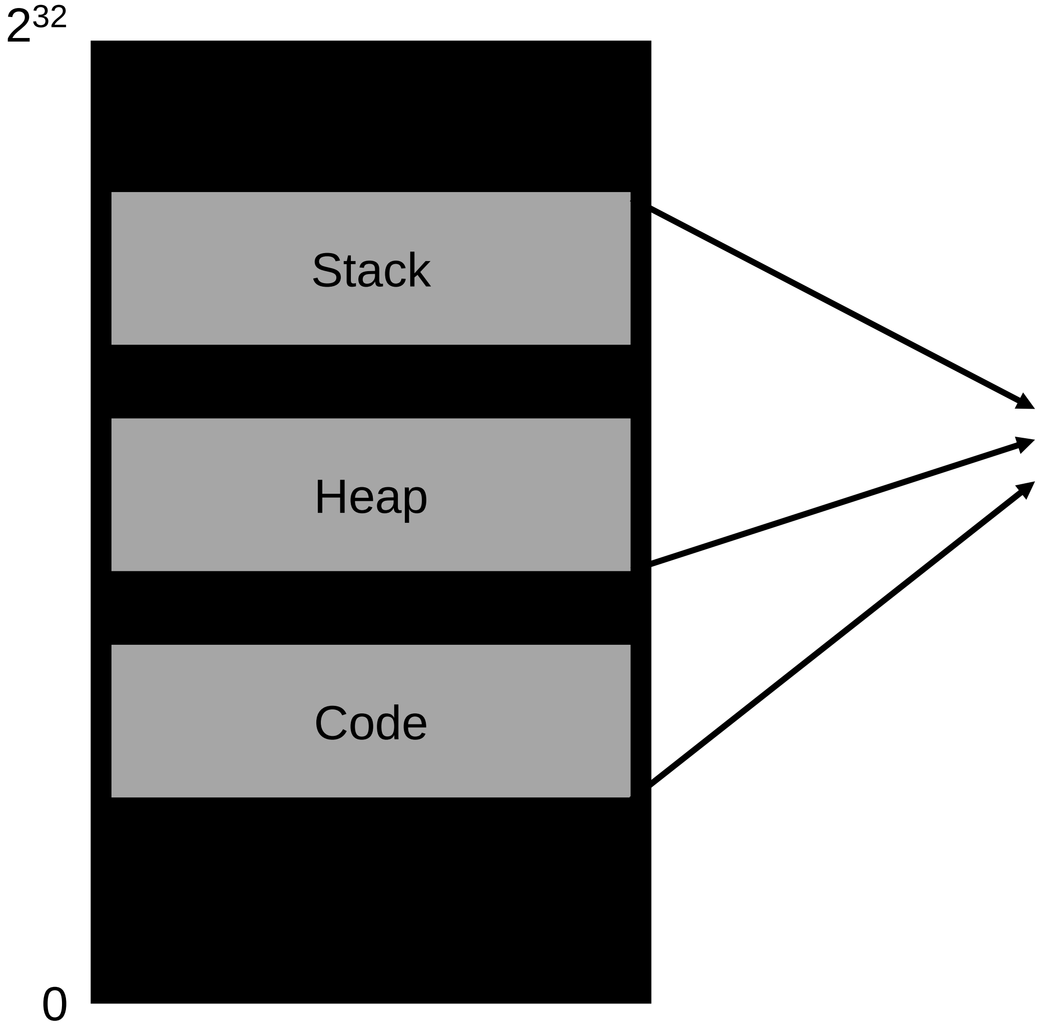
World with ASLR

Enable ASLR by:

```
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

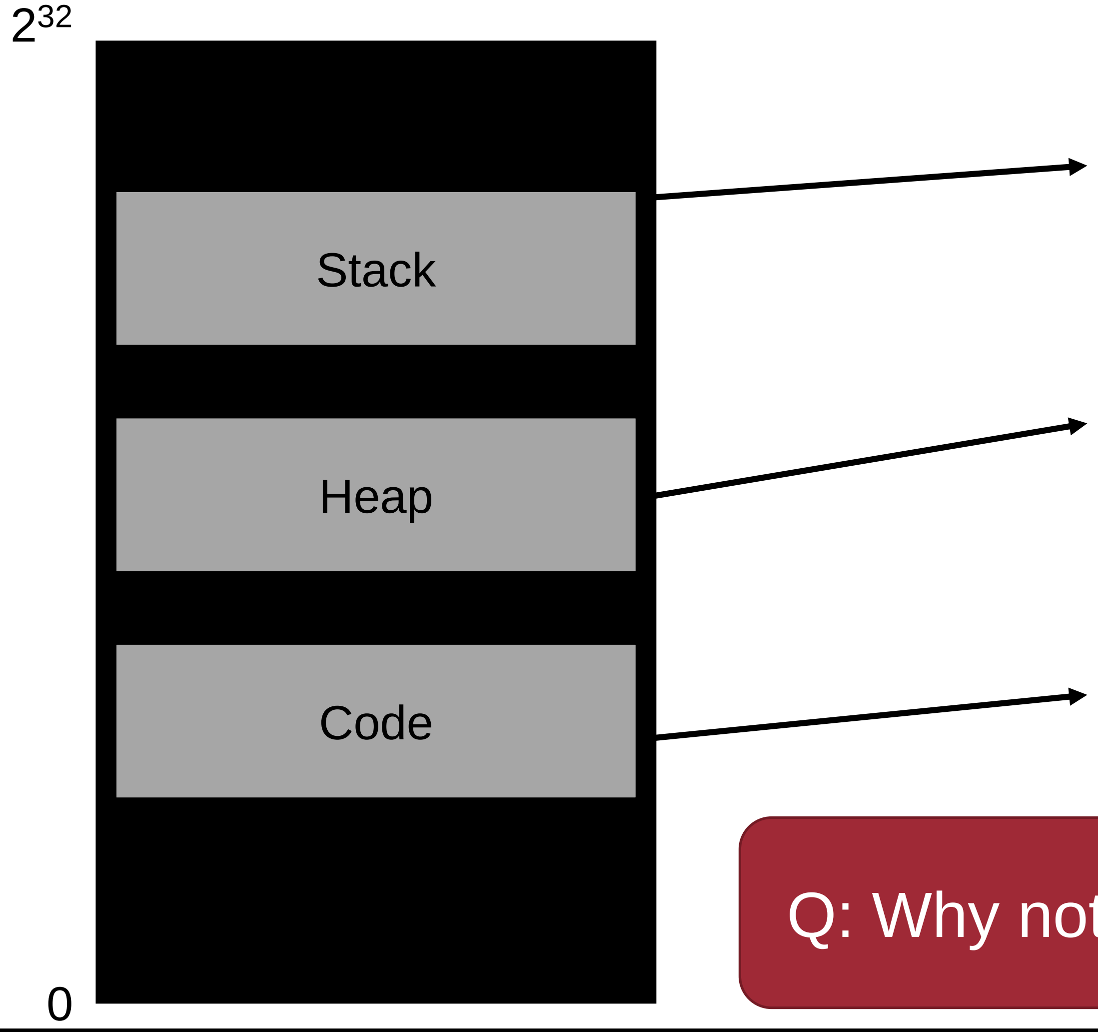
Why 2? How would you figure out the meaning of this parameter?

ASLR



Randomized based addresses

Randomness of ASLR on Linux x86



24 bits of randomness

16 bits of randomness

16 bits of randomness

Q: Why not fully utilize 32 bits for randomization?

Previous Exploits Will *Not* Work w/ ASLR

- Memory layout will be ***randomized*** with ASLR.
  - Randomizes the ***base address*** of the stack, heap, and code segments

- We cannot know the address of shellcode nor library functions.

Are we safe now?

ASLR Entropy is Small on x86

- Just 16 bits (heap, libraries) on x86
- Brute-forcing is possible for server applications that use ***fork***ing.
  - Forked process has the same address space layout as its parent
  - Once we know the address of a function in LIBC, we can deduce the addresses of all functions in LIBC!

- Reference:
  On the Effectiveness of Address-Space Randomization,
  CCS 2004

The Attack

- Target: Apache web server
  - Forks children on requests

- Vulnerability: Buffer Overflow

- Method: Return to LIBC (`usleep`)
  - Try to brute-force the address of `usleep`
  - The fake parameter of `usleep` is 16,000,000 (waiting for 16 sec.)

- Once we know the address of `usleep`, we can determine the address of `exec` or `system`

Defense #3: Canary

Canary in a Cole Mine

Mitigating Buffer Overflows with Canary

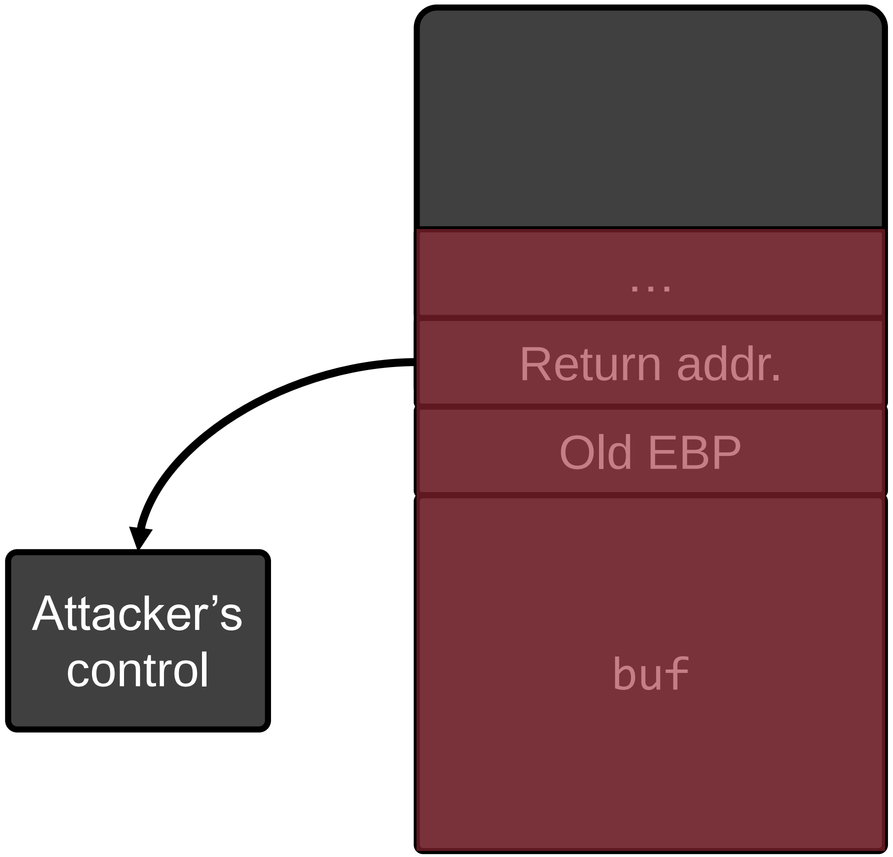
- First introduced in 1998.

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, ***USENIX Security 1998***

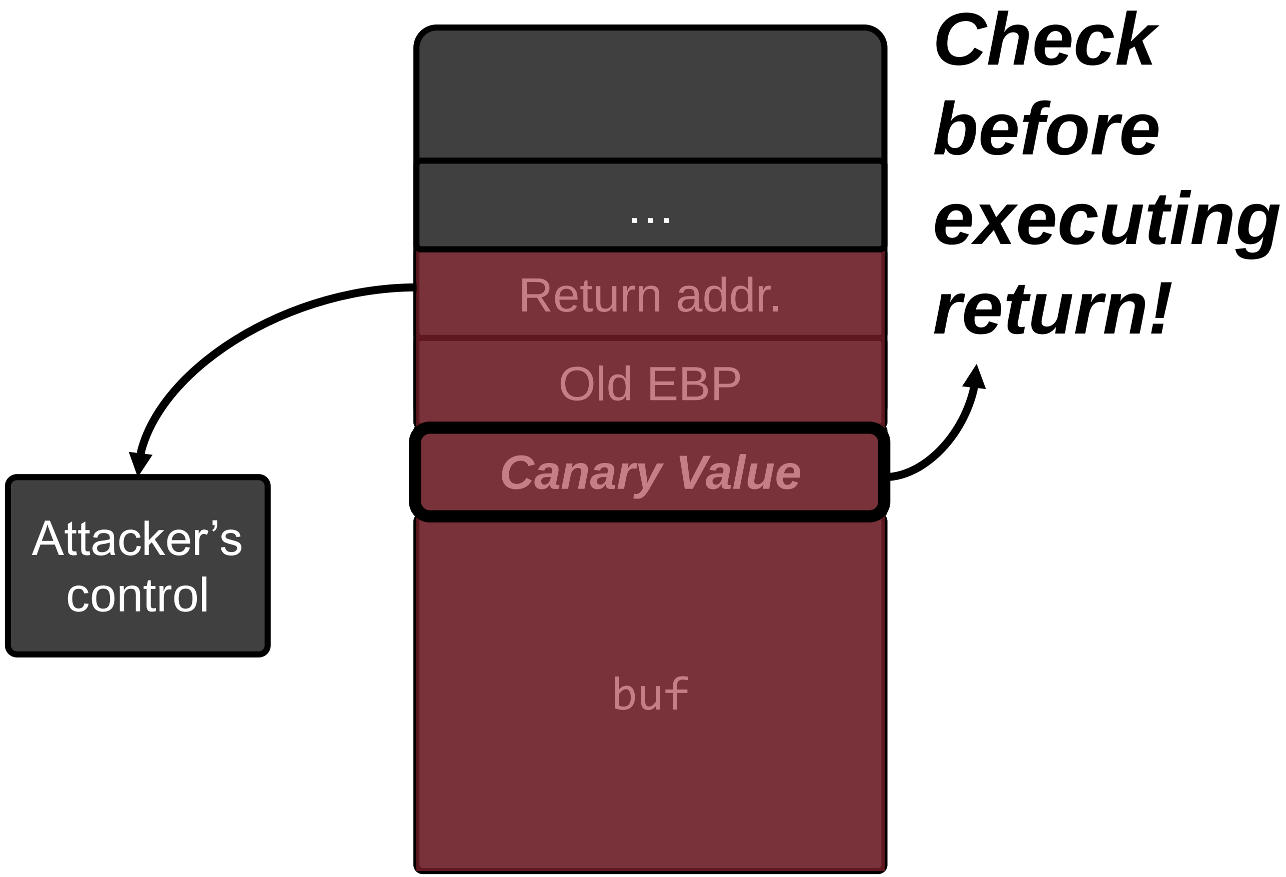
- Similar approach called ***StackShield*** was introduced in 1999.

- Not necessarily used for stack, but can also be used for heap

Stack Canary (a.k.a. Stack Cookie)



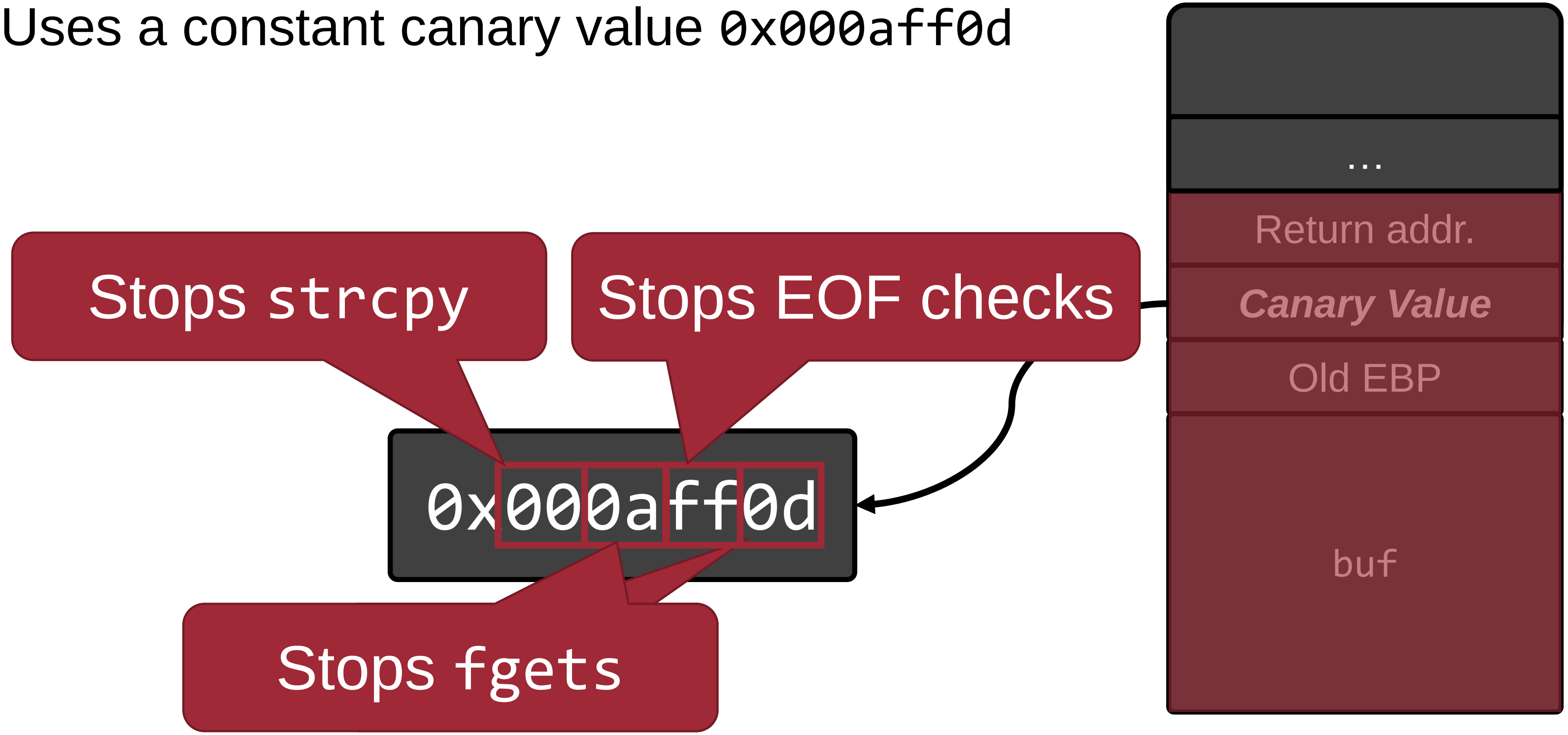
Without Stack Canary



With Stack Canary

StackGuard (1998)

Uses a constant canary value `0x000aff0d`

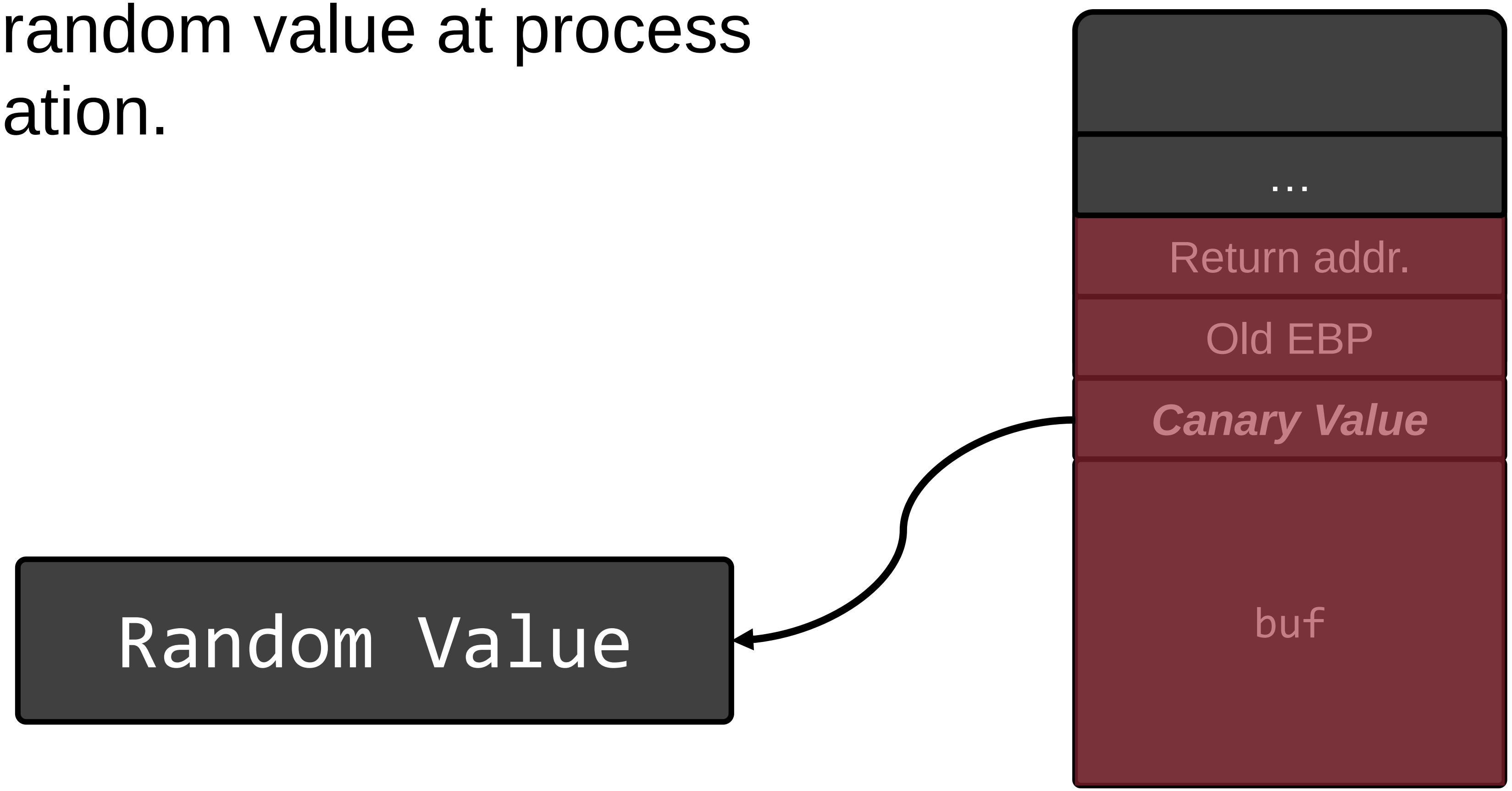


Problem of Constant Canary Value

`memcpy?`

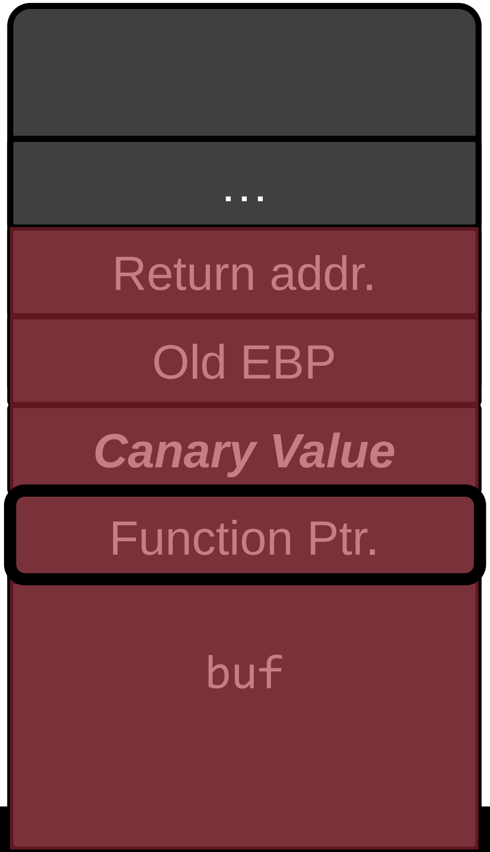
Random Canaries

Pick a random value at process initialization.



Problems Still Exist

Local variables are not protected!



Reordering Local Variables

Always put local buffers after local pointers.

(This idea is implemented in GCC 4.1 in 2005.)

```
80483fb: push   ebp
80483fc: mov    ebp,esp
80483fe: sub    esp,0x100
8048404: push   DWORD PTR [ebp+0x8]
8048407: lea    eax,[ebp-0x100]
804840d: push   eax
804840e: call   80482d0 <strcpy@plt>
8048413: add    esp,0x8
8048416: leave
8048417: ret
```

```
804844b: push   ebp
804844c: mov    ebp,esp
804844e: sub    esp,0x108
8048454: mov    eax,DWORD PTR [ebp+0x8]
8048457: mov    DWORD PTR [ebp-0x108],eax
804845d: mov    eax,gs:0x14
8048463: mov    DWORD PTR [ebp-0x4],eax
8048466: xor    eax,eax
8048468: push   DWORD PTR [ebp-0x108]
804846e: lea    eax,[ebp-0x104]
8048474: push   eax
8048475: call   8048320 <strcpy@plt>
804847a: add    esp,0x8
804847d: mov    eax,DWORD PTR [ebp-0x4]
8048480: xor    eax,DWORD PTR gs:0x14
8048487: je     804848e <somefn+0x43>
8048489: call   8048310 <__stack_chk_fail@plt>
804848e: leave
804848f: ret
```

Without Stack Canary

`gcc -fno-stack-protector`

With Stack Canary

`gcc -fstack-protector`

Random canary value at `gs:0x14`

Why?

```
804844b: push   ebp
804844c: mov    ebp,esp
804844e: sub    esp,0x108
8048454: mov    eax,DWORD PTR [ebp+0x8]
8048457: mov    DWORD PTR [ebp-0x108],eax
804845d: mov    eax,gs:0x14
8048463: mov    DWORD PTR [ebp-0x4],eax
8048466: xor    eax,eax
8048469: push   DWORD PTR [ebp-0x108]
804846e: lea    eax,[ebp-0x104]
8048474: push   eax
8048475: call   8048320 <strcpy@plt>
804847a: add    esp,0x8
804847d: mov    eax,DWORD PTR [ebp-0x4]
8048480: xor    eax,DWORD PTR gs:0x14
8048487: je     804848e <somefn+0x43>
8048489: call   8048310 <__stack_chk_fail@plt>
804848e: leave
804848f: ret
```

With Stack Canary

`gcc -fstack-protector`

GS* Segment Register?

- x86 maintains a Local Descriptor Table (LDT) in memory.
- Segment registers hold an offset of the LDT.
- On Linux, GS* segment register points to an entry of LDT, which represents Thread Control Block (TCB).

```
typedef struct {
  void *tcb;              /* gs:0x00 Pointer to the TCB. */
  dtv_t *dtv;             /* gs:0x04 */
  void *self;             /* gs:0x08 Pointer to the thread descriptor.  */
  int multiple_threads;   /* gs:0x0c */
  uintptr_t sysinfo;      /* gs:0x10 Syscall interface */
  uintptr_t stack_guard;  /* gs:0x14 Random value used for stack protection */
  uintptr_t pointer_guard;/* gs:0x18 Random value used for pointer protection */
  int gscope_flag;        /* gs:0x1c */
  int private_futex;      /* gs:0x20 */
  void *__private_tm[4];  /* gs:0x24 Reservation of some values for the TM ABI.  */
  void *__private_ss;     /* gs:0x34 GCC split stack support.  */
} tcbhead_t;
```


* On x86-64, FS is used

Who Initializes [gs:0x14]?

Runtime Dynamic Linker (RTLD) does it every time it launches a process

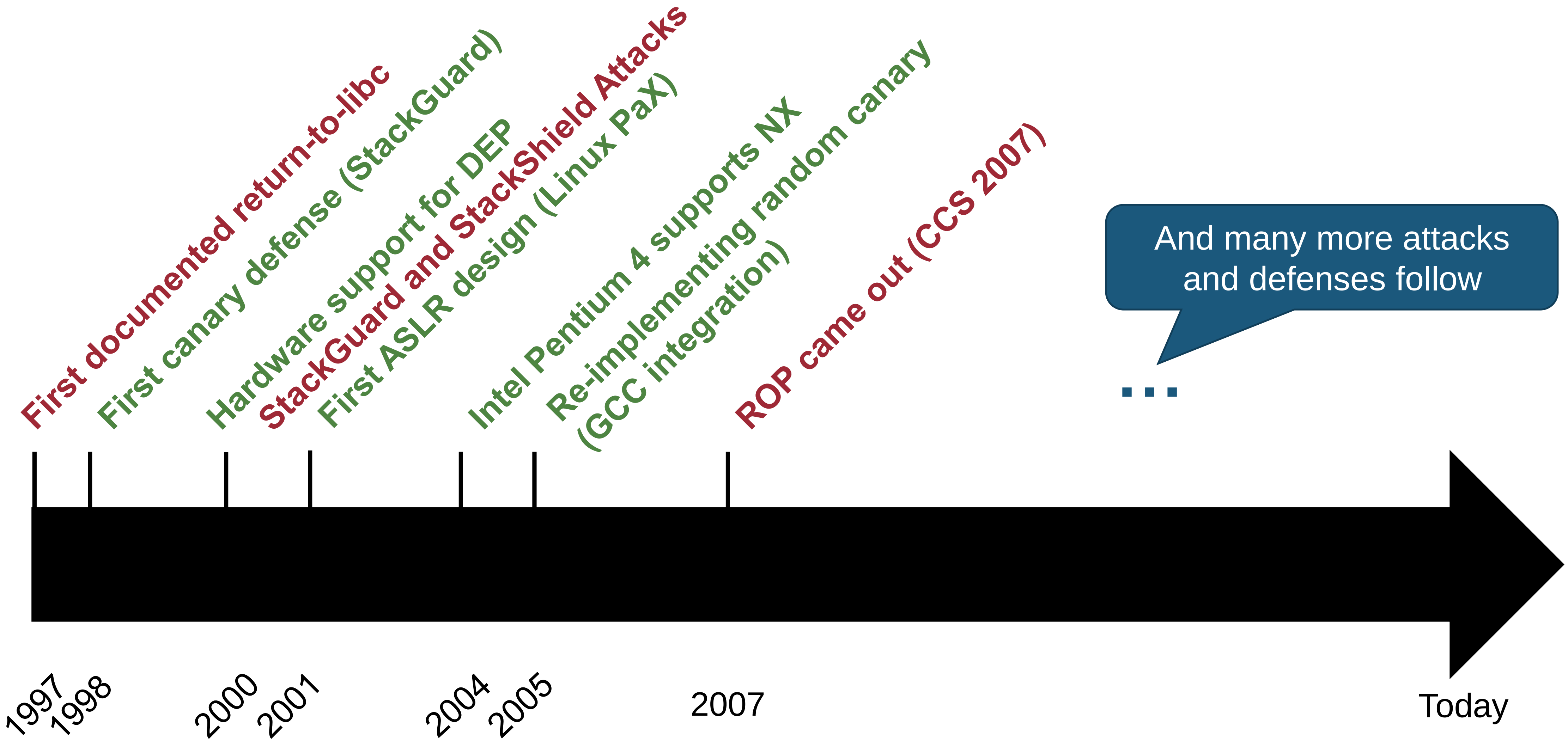
```
// Below is roughly what RTLD does at process creation time
uintptr_t ret;
int fd = open("/dev/urandom", O_RDONLY);
if (fd >= 0) {
   ssize_t len = read(fd, &ret, sizeof(ret));
   if (len == (ssize_t) sizeof(ret)) {
      // inlined assembly for moving ret to [gs:0x14]
   }
}
```

GCC ProPolice Implementation

- Uses random canary values for every process creation
- Puts buffers after any local pointers on the stack

```
804844b: push   ebp
804844c: mov    ebp,esp
804844e: sub    esp,0x108
8048454: mov    eax,DWORD PTR [ebp+0x8]
8048457: mov    DWORD PTR [ebp-0x108],eax
804845d: mov    eax,gs:0x14
8048463: mov    DWORD PTR [ebp-0x4],eax
8048466: xor    eax,eax
8048468: push   DWORD PTR [ebp-0x108]
804846e: lea    eax,[ebp-0x104]
8048474: push   eax
8048475: call   8048320 <strcpy@plt>
804847a: add    esp,0x8
804847d: mov    eax,DWORD PTR [ebp-0x4]
8048480: xor    eax,DWORD PTR gs:0x14
8048487: je     804848e <somefn+0x43>
8048489: call   8048310 <__stack_chk_fail@plt>
804848e: leave
804848f: ret
```

Summary


First documented return-to-libc
First canary defense (StackGuard)
Hardware support for DEP
StackGuard and StackShield Attacks
First ASLR design (Linux PaX)
Intel Pentium 4 supports NX
Re-implementing random canary
(GCC integration)
ROP came out (CCS 2007)
And many more attacks and defenses follow
...
1997
1998
2000
2001
2004
2005
2007
Today

Questions?