Stack protection

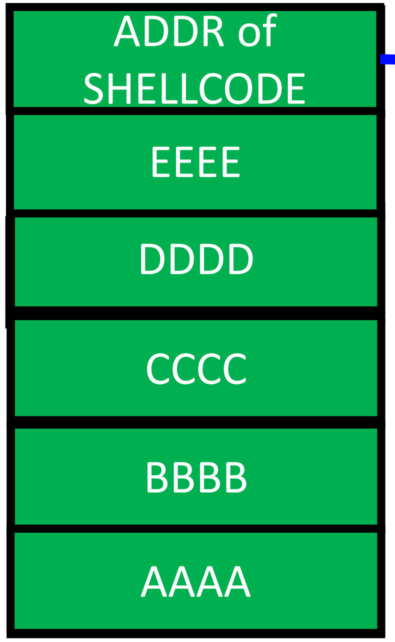
Insu Yun

Most of materials from CS419/579 Cyber Attacks & Defense in OSU

Today's lecture

- Understand spatial memory safety
- Understand SoftBound
- Understand stack cookie
- Understand weakness of stack cookie

Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

How to defend against stack overflow?

- Prevent buffer overflow!
  - A direct defense
  - Could be accurate but could be slow..

Softbound, etc.

- Make exploit hard!
  - An indirect defense
  - Could be inaccurate but could be fast..

Exploit Mitigation
Stack cookie, DEP, ASLR, etc.

Softbound: Bound checking for C!

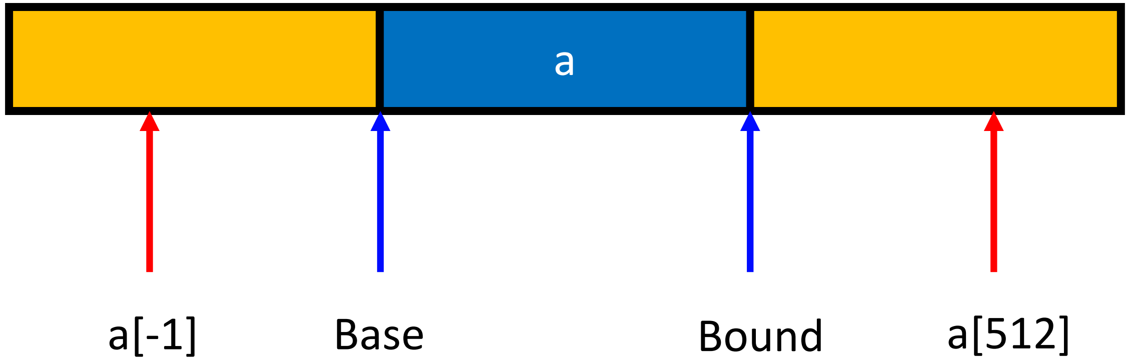
In Proceedings of
Programming Language Design and Implementation
(PLDI) 2009

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic
Computer and Information Sciences Department, University of Pennsylvania
Technical Report MS-CIS-09-01 — January 2009

Memory Safety = Temporal Safety (e.g., use-after-free)
+ Spatial Safety (e.g., buffer overflow)

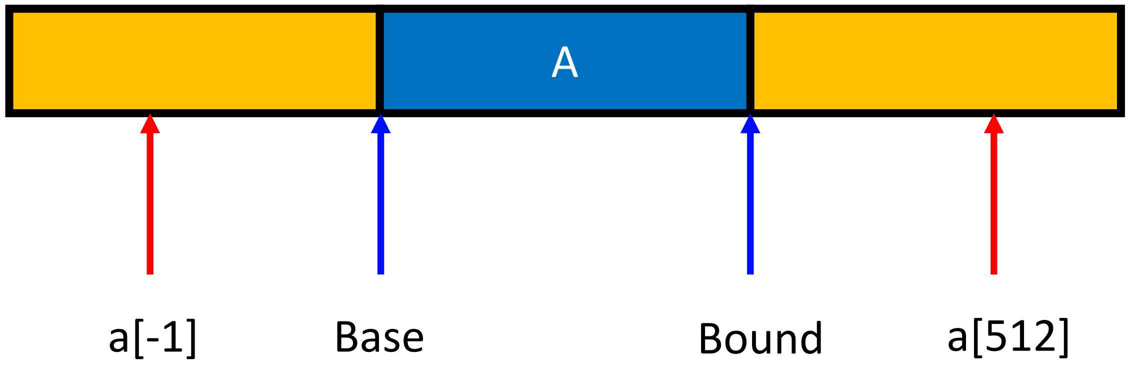
Spatial safety



- Guarantee that an access does not go
  1) behind the Base and
  2) over the Bound

Softbound: Bounds checking

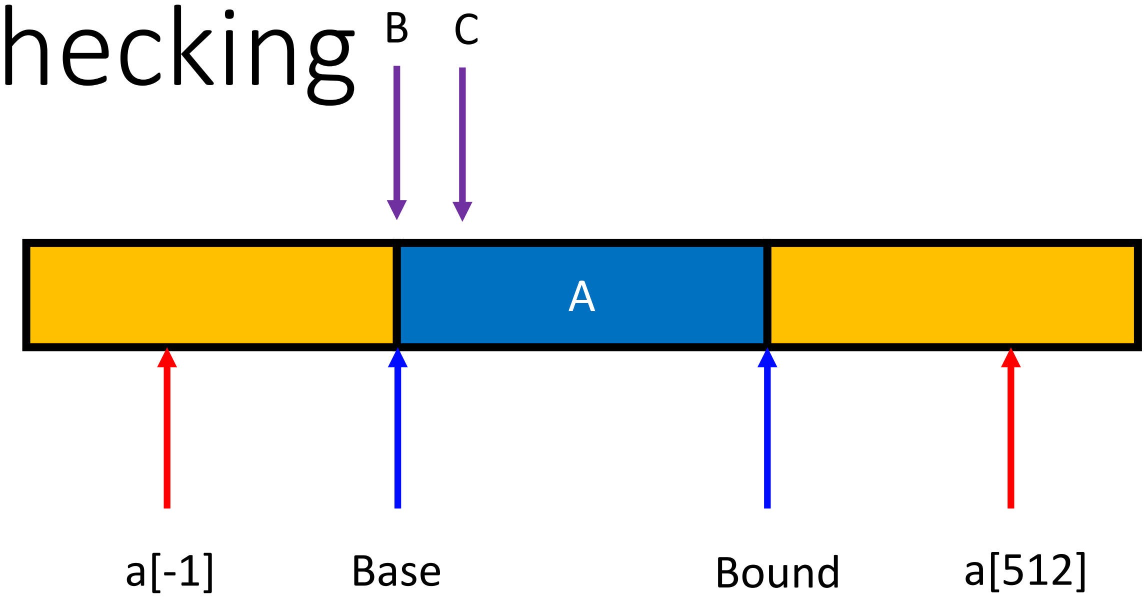
- A FAT pointer
  - `char *a`
    - char *a_base;
    - char *a_bound;



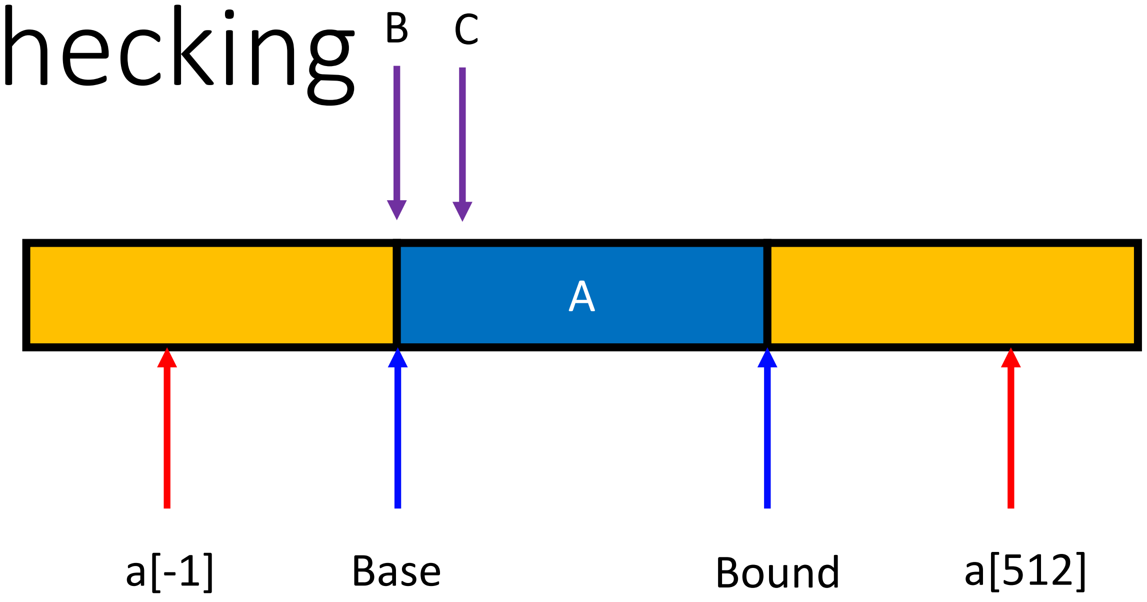
- Allocation
  - `a = (char*)malloc(512)`
    - a_base = a;
    - a_bound = a+512
  - Access must be between [a_base, a_bound)
    - a[0], a[1], a[2], …, and a[511] are OK
    - a[512] NOT OK
    - a[-1] NOT OK

Softbound: Bounds checking

- Propagation
  - `char *b = a;`
    - b_base = a_base;
    - b_bound = a_bound;
  - `char *c = &b[2];`
    - c_base = b_base;
    - c_bound = b_bound;



Softbound: Bounds checking



- Propagation
  - `char *c = &b[2];`
    - c_base = b_base;
    - c_bound = b_bound;
  - `c[1] = 'a';`
    - c== b+2 == a+2
    - c+1 == b+3 == a+3
    - c_base <= c+1 && c+1 < c_bound
  - `c[510] = 'a';`
    - c == b+2 == a+2
    - c+510 == b+510+2 == a+510+2 == a+512
    - c_base <= c+510 but c+510 >= c_bound
    - Disallow write!

Softbound: Bounds checking

- Buffer?
  - `strcpy(c, “A”*510)`
- When copying 510th character:
  - `c[510] = ’A’;`
    - c+510 > c_bound (c+510 == a+512 > bound…)
    - Detect buffer overrun!

- This is how Java and other languages (e.g., rust) protect buffer overrun
- Even for `std::vector` in C++

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic
Computer and Information Sciences Department, University of Pennsylvania
Technical Report MS-CIS-09-01 — January 2009

In Proceedings of
Programming Language Design and Implemenetation
(PLDI) 2009

```
ptr = malloc(size);
ptr_base = ptr;
ptr_bound = ptr + size;
if (ptr == NULL) ptr_bound = NULL;

int array[100];
ptr = &array;
ptr_base = &array[0];
ptr_bound = &array[100];

newptr = ptr + index;     // or &ptr[index]
newptr_base = ptr_base;
newptr_bound = ptr_bound;
```

Drawbacks

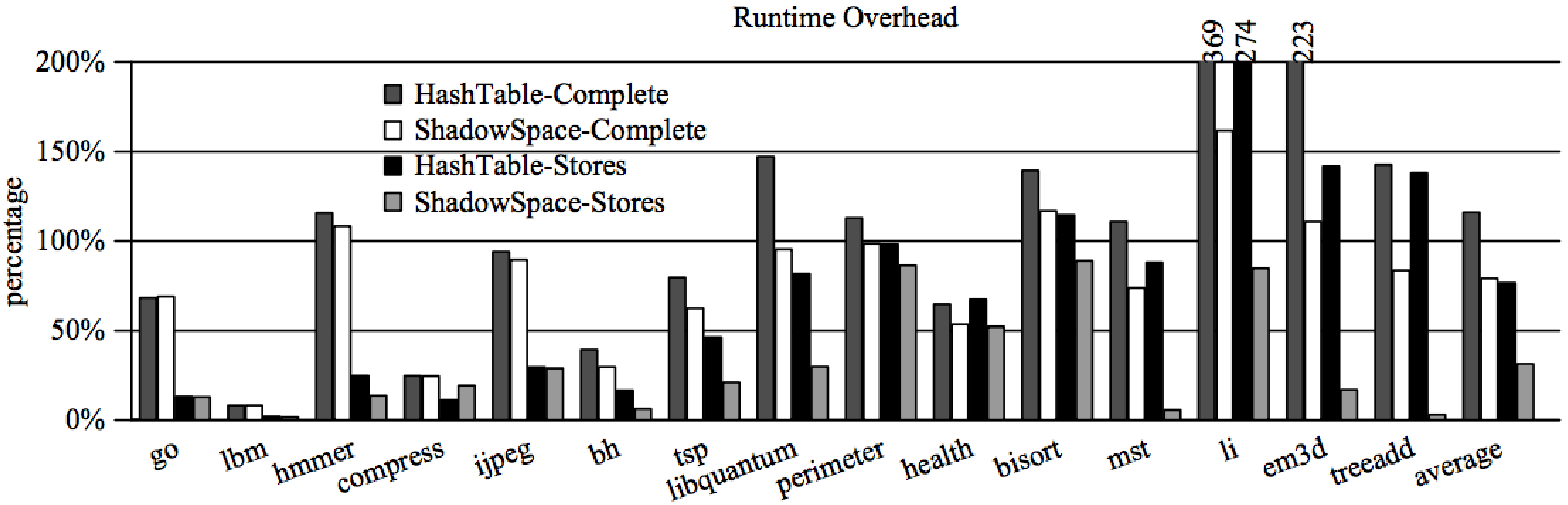
- +2x overhead on storing a pointer
  - char *a
    - char *a_base;
    - char *a_bound;
- +2x overhead on assignment
  - char *b = a;
    - b_base = a_base;
    - b_bound = a_bound;
- +2 comparisons added on access
  - c[i]
    - if(c+i >= c_base)
    - if(c+i < c_bound)

Many other problems…
Use more cache
More TLBs
etc.…

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic

Computer and Information Sciences Department, University of Pennsylvania



Security vs. Performance

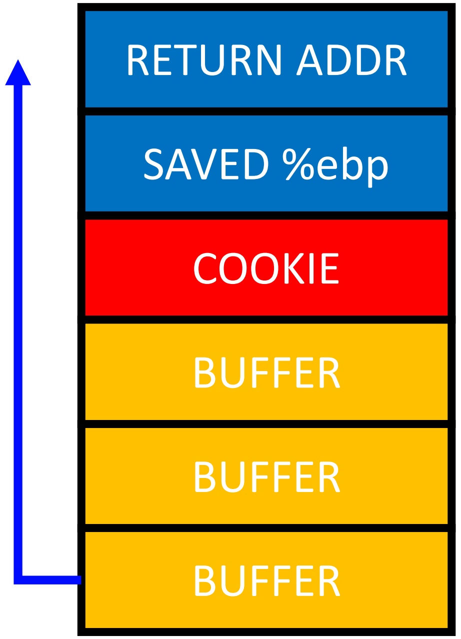
- 100% Buffer Overflow Free
  - You pay +200% Performance Overhead
  - Think about the economy…

An Economic Defense: Stack Cookie

- A defense specific to ***sequential*** stack overflow

- On a function call
  - cookie = some_random_value

- Before the function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);

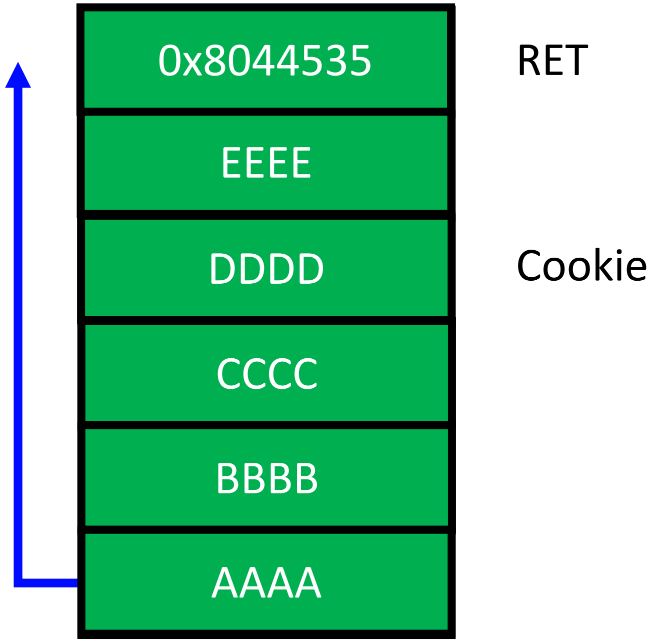


Stack Cookie: Attack Example

- strcpy(buffer, “AAAABBBBCCCCDDDDEEEE\x35\x45\x04\x08”)

- On a function call
  - cookie = some_random_value

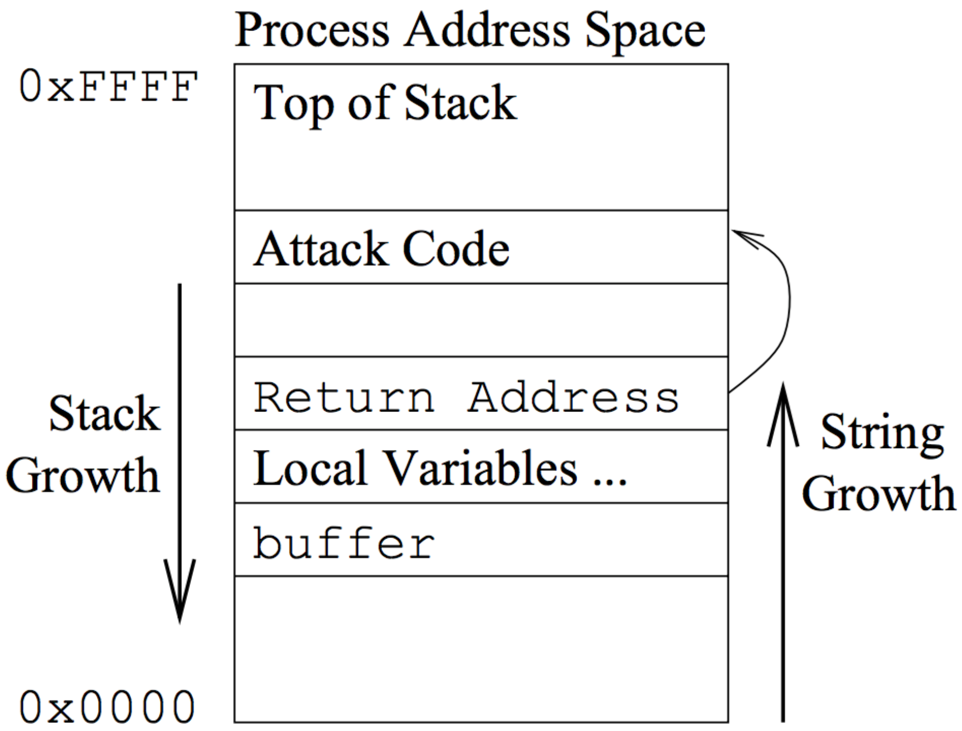
- Before a function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);

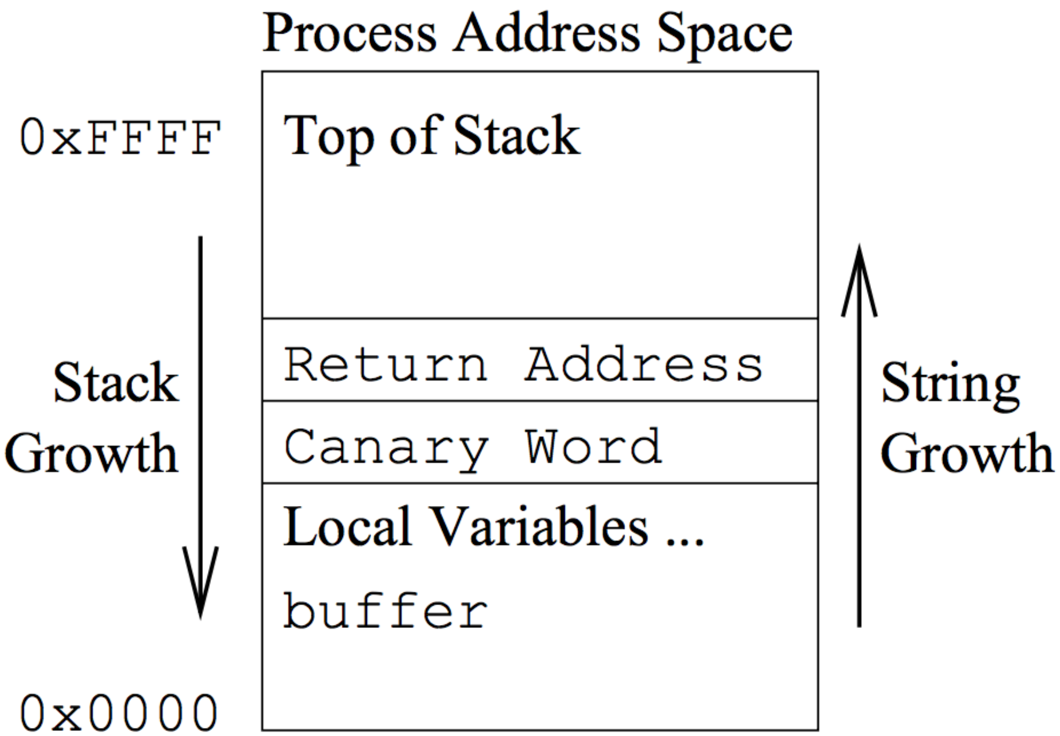


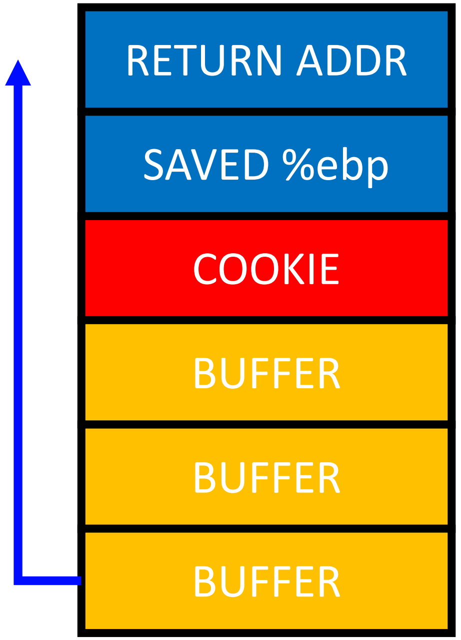
StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*

Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton,[†] Jonathan Walpole,
Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
immunix-request@cse.ogi.edu, http://cse.ogi.edu/DISC/projects/immunix

In Proceedings of
The 7th USENIX Security Symposium (1998)







Stack Cookie

GCC ProPolice

```c
void input_func() {
  char buf[20];
  scanf("%s", buf);
  printf("%s\n", buf);
}
```

```
gcc -o a a.c -m32
```

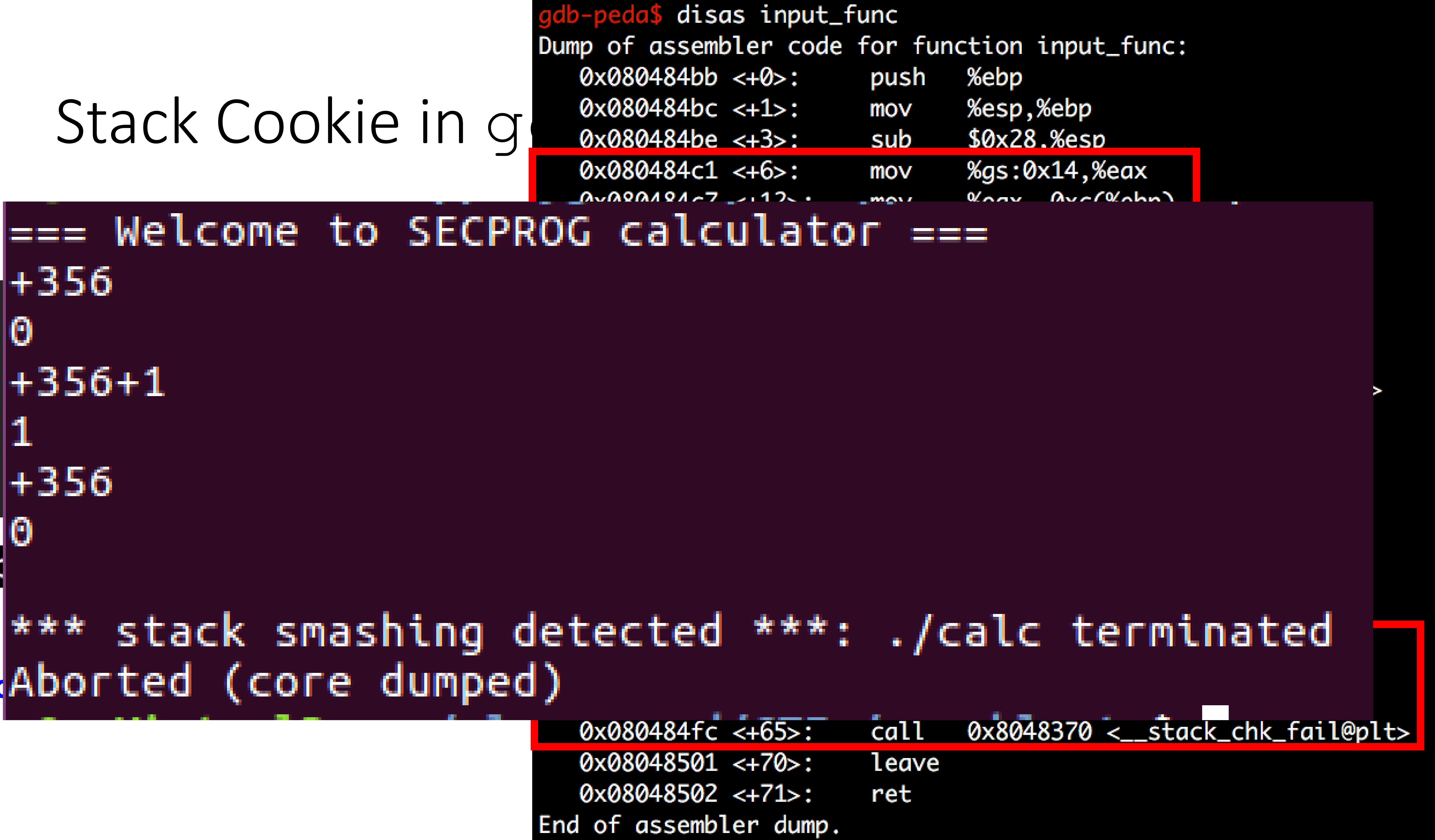
Cookie stored in -0xc(%ebp)

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
   0x080484ca <+15>:    xor    %eax,%eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax
   0x080484f3 <+56>:    xor    %gs:0x14,%eax
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Get canary from %gs
Store canary at ebp-c
Clear canary in %eax

Get canary in stack
Xor that with value in %gs

Stack Cookie in g



```c
// @glibc/sysdeps/i386/nptl/tls.h
typedef struct
{
  void *tcb;                /* Pointer to the TCB.  Not necessarily the
                               thread descriptor used by libpthread.  */
  dtv_t *dtv;
  void *self;                /* Pointer to the thread descriptor.  */
  int multiple_threads;
  uintptr_t sysinfo;
  uintptr_t stack_guard;
  uintptr_t pointer_guard;
  int gscope_flag;
  /* Bit 0: X86_FEATURE_1_IBT.
     Bit 1: X86_FEATURE_1_SHSTK.
   */
  unsigned int feature_1;
  /* Reservation of some values for the TM ABI.  */
  void *__private_tm[3];
  /* GCC split stack support.  */
  void *__private_ss;
  /* The lowest address of shadow stack,  */
  unsigned long ssp_base;
} tcbhead_t;
```

https://tc.gts3.org/cs6265/2022/_static/tut.pdf

Stack Cookie: Overhead

- 2 memory move
  - +1 for store, +1 for read
- 1 compare

- Per each function call

- 1~5% overhead

Benchmark:
SPECint, SPECfloat

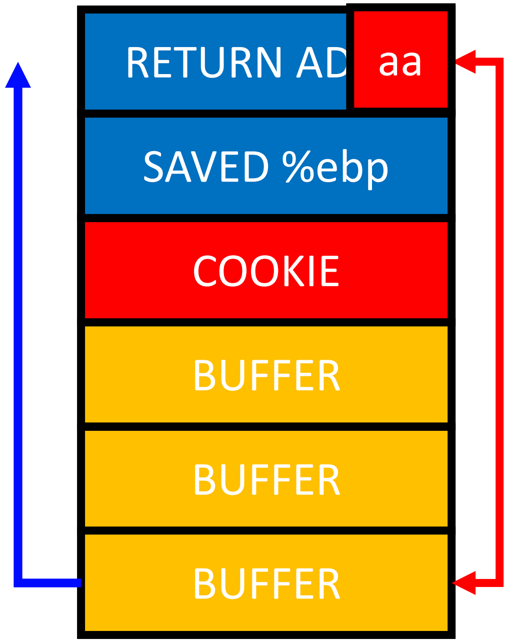
Compile Options	CINT		CFP	
-fno-stack-protector_-m32	257		107	
-fstack-protector-all_-m32	268	(104.28%)	113	(105.61%)

Stack Cookie: Weaknesses

- Effective for common mistakes
  - strcpy/memcpy
  - read/scanf
  - Missing bound check in a for loop

- But can only block sequential overflow

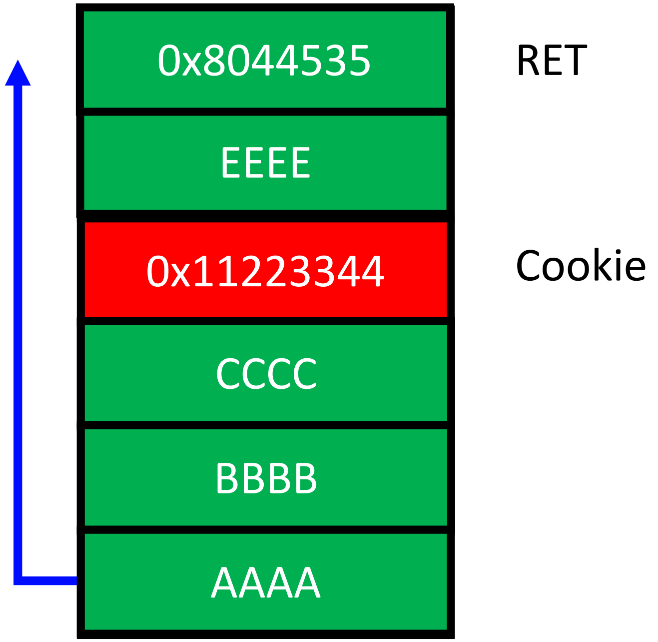
- What if buffer[24] = 0xaa?



Stack Cookie: Weaknesses

- Fail if attacker can guess the cookie value
  - strcpy(buf, “AAAABBBBCCCC\x44\x33\x22\x11EEEE…”)
  - (stack-cookie-1)

- -> Use a random value for a cookie!
  - Is rand() safe?
- See https://www.includehelp.com/c-programs/guess-a-random-number.aspx



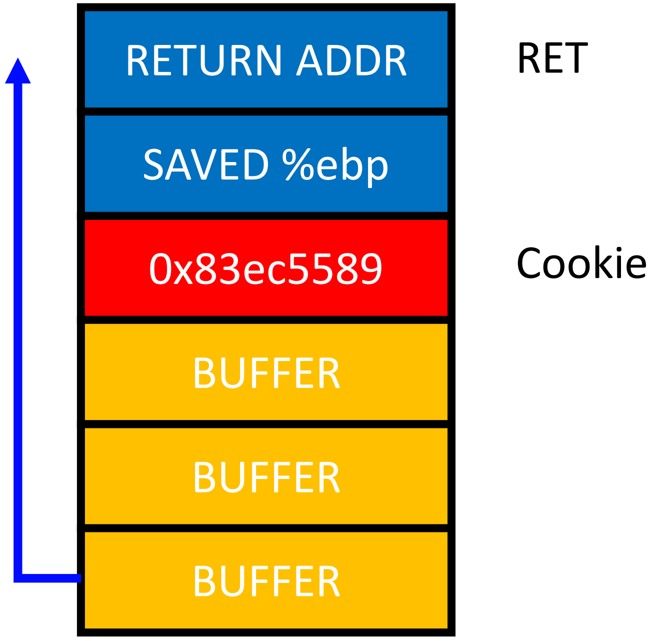
Stack Cookie: Weaknesses

- Security in 32-bit Random Cookie
  - One chance over 2^{32} (4.2 billion) trial
  - Seems super secure!

- Fail if attacker can read the cookie value…

```
0x080484c1 <+6>:     mov    %gs:0x14,%eax
0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
0x080484ca <+15>:    xor    %eax,%eax
```

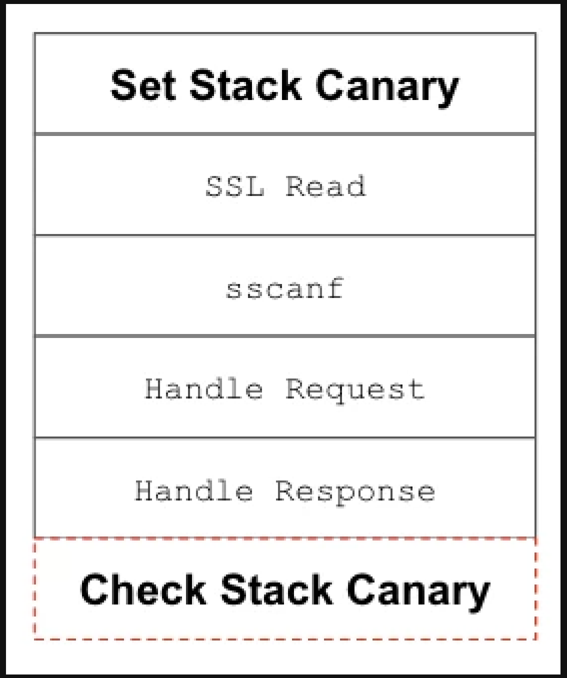
  - Maybe you can’t read %gs:0x14
  - But, what about -0xc(%ebp)?



Stack Cookie: Weaknesses

- Check when we return

  -> Do something bad before return



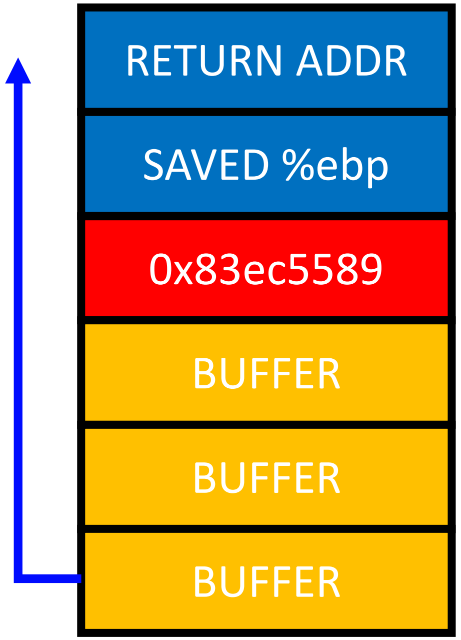
Stack layout and calls.

Stack Cookie: Weaknesses

- Random becomes non-random if fork()-ed..

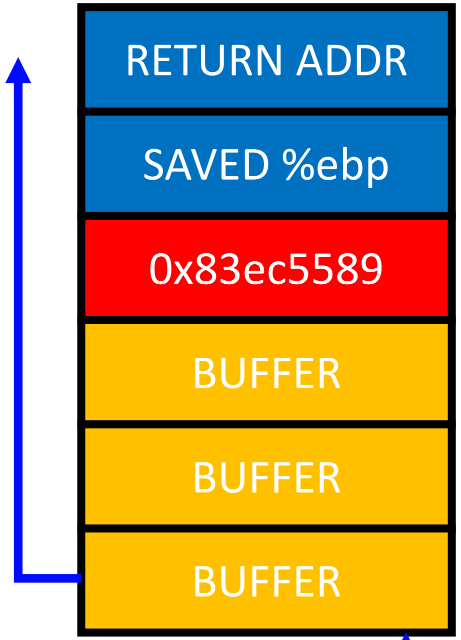
Stack Cookie: Weaknesses

- Servers…



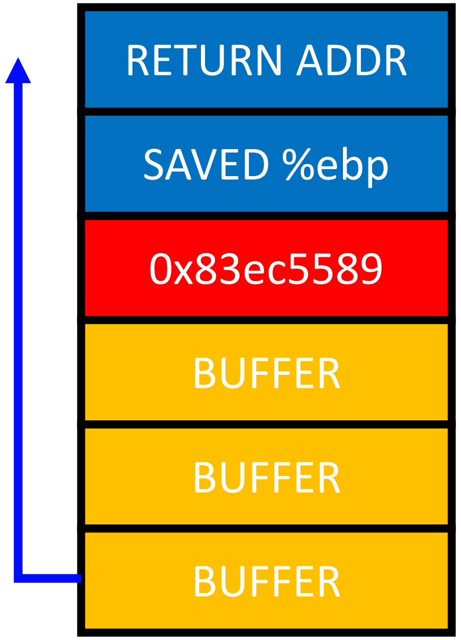
fork()!

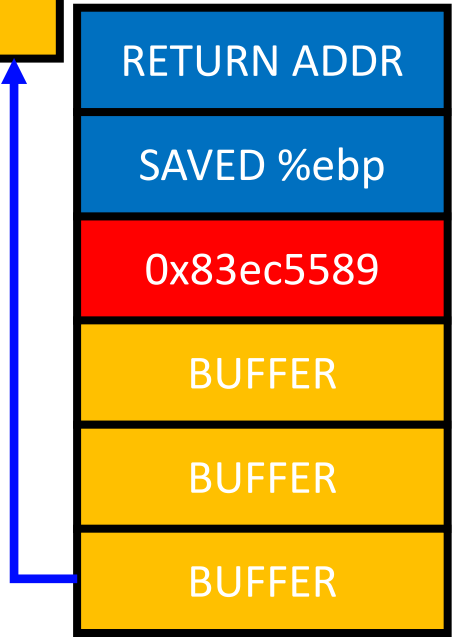
Why?



fork()!

fork()!





Stack Cookie: Bypassing ProPolice

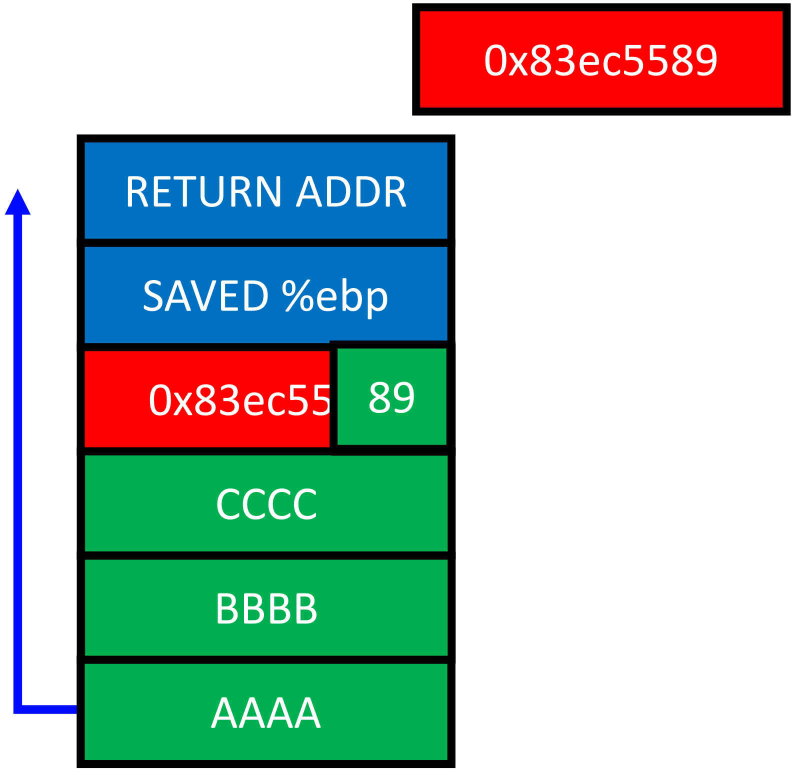
- Assumption
  - A server program contains a sequential buffer overflow vulnerability
  - A server program uses `fork()`
  - A server program let the attacker know if it detected stack smashing or not
    - E.g., an error message, “stack smashing detected”, etc.

```
=== Welcome to SECPROG calculator ===
+356
0
+356+1
1
+356
0

*** stack smashing detected ***: ./calc terminated
Aborted (core dumped)
```

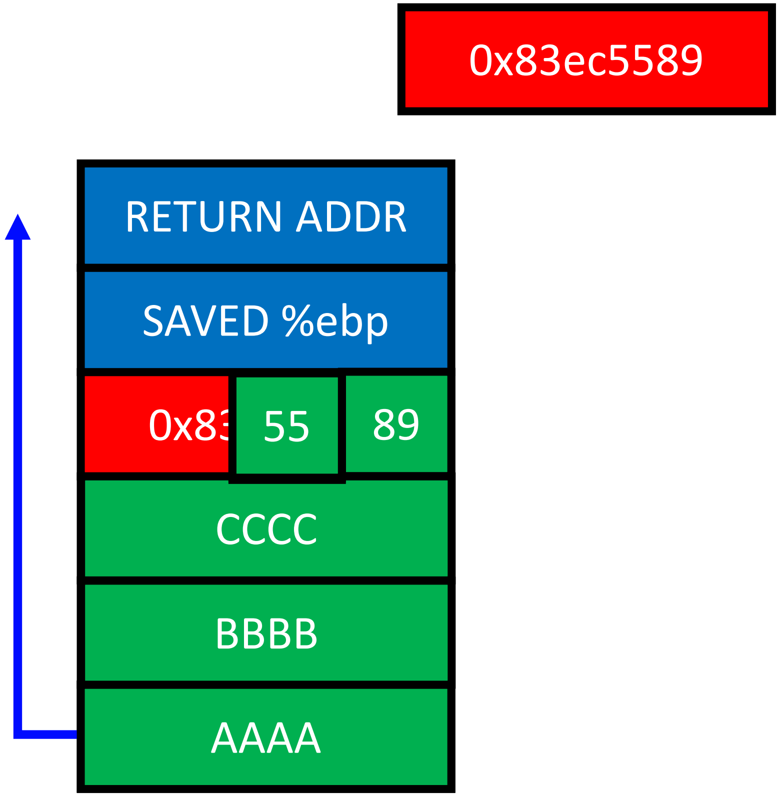
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess only the last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x88
  - When testing 0x89
    - No smashing and return correctly



Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x54
  - When testing 0x55
    - No smashing and return correctly



Stack Cookie: Bypassing ProPolice

- An easy brute force attack
  - Max 256 trials to match 1 byte value
  - Move forward if found the value
    - In 32-bit: 4 * 256 = max 1,024 trials
    - In 64-bit: 8 * 256 = max 2,048 trials

Stack Cookie: Weaknesses

- Random becomes non-random if fork()-ed..

CVE-2013-2028: nginx stack buffer overflow

```
static ngx_int_t
ngx_http_read_discarded_request_body(ngx_http_request_t *r)
{
    size_t     size;
    ssize_t    n;
    ngx_int_t  rc;
    ngx_buf_t  b;
    u_char     buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
    ...
        size = (size_t) ngx_min(r->headers_in.content_length_n,
                                NGX_HTTP_DISCARD_BUFFER_SIZE);

        n = r->connection->recv(r->connection, buffer, size);
    ...
    }
}
```

- Exploitation on x64:
  - The problem of stack cookie/carnary can be overcome easily by brute-forcing byte by byte. If we send an extra byte and a worker process crashes, it will return nothing thus we know our cookie value is wrong, we try another value until we receive some output.
  - Then we need to bypass ASLR and DEP. The exploitation for 32-bit in the metasploit module won't work, since it will bruteforce the libc address and it's not feasible given the large address space in x64.