

CSE 410/510 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

Location: Obrian 109

Time: Monday, Wednesday 5:00PM-6:20PM

Course Evaluation

Begins: 3/6/2022

Ends: 3/13/2022

If 90% of student submit the evaluation, all of the class will get **10** bonus points.

44 students. So 40 **evaluations!!**

Midterm Written Exam and CTF

3/14/2022 and 3/16/2022 in class. **Must be in-person.**

3 hours in total.

Last and This Class

1. Stack-based buffer overflow
 - a. Defense.

Attacker's Goal

Take control of the victim's machine

- Hijack the execution flow of a running program
- Execute arbitrary code

Requirements

- Inject attack code or attack parameters
- Abuse vulnerability and modify memory such that control flow is redirected

Change of control flow

- ***alter a code pointer*** (RET, function pointer, etc.)
- change memory region that should not be accessed

Overflow Types

Overflow some ***code pointer***

- Overflow memory region on the stack
 - overflow function return address
 - overflow function frame (base) pointer
 - overflow longjmp buffer
- Overflow (dynamically allocated) memory region on the heap
- Overflow function pointers
 - stack, heap, BSS

Other pointers?

Can we exploit other pointers as well?

1. Memory that is used in a **value** to influence mathematical operations, conditional jumps.
2. Memory that is used as a **read pointer** (or offset), allowing us to force the program to access arbitrary memory.
3. Memory that is used as a **write pointer** (or offset), allowing us to force the program to overwrite arbitrary memory.
4. Memory that is used as a **code pointer** (or offset), allowing us to redirect program execution!

Typically, you use one or more vulnerabilities to achieve multiple of these effects.

Defenses

- Prevent buffer overflow
 - A direct defense
 - Could be accurate but could be slow
 - Good in theory, but not practical in real world
- Make exploit harder
 - An indirect defense
 - Could be inaccurate but could be fast
 - Simple in theory, widely deployed in real world

Examples

- Base and bound check
 - Prevent buffer overflow!
 - A direct defense
- Stack Cookie
 - An indirect defense
 - Prevent overwriting return address
- Data execution prevention (DEP, NX, etc.)
 - An indirect defense
 - Prevent using of shellcode on stack

Spatial Memory Safety – Base and Bound check

```
char *a
```

- char *a_base;
- char *a_bound;

```
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

Spatial Memory Safety – Base and Bound check

Propagation

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

Overhead - Based and Bound

+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)`

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

santoshn@cis.upenn.edu jianzhou@cis.upenn.edu milom@cis.upenn.edu stevez@cis.upenn.edu

Abstract

The serious bugs and security vulnerabilities facilitated by C/C++'s lack of bounds checking are well known, yet C and C++ remain in widespread use. Unfortunately, C's arbitrary pointer arithmetic,

address on the stack, address space randomization, non-executable stack), vulnerabilities persist. For one example, in November 2008 Adobe released a security update that fixed several serious buffer overflows [2]. Attackers have reportedly exploited these buffer-overflow vulnerabilities by using banner ads on websites to redi-

HardBound: Architectural Support for Spatial Safety of the C Programming Language

Joe Devietti *

University of Washington
devietti@cs.washington.edu

Colin Blundell

University of Pennsylvania
blundell@cis.upenn.edu

Milo M. K. Martin

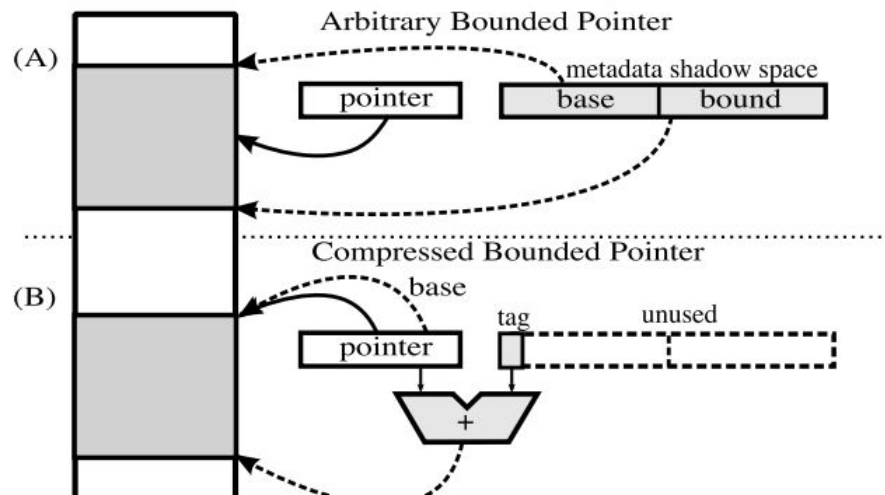
University of Pennsylvania
milom@cis.upenn.edu

Steve Zdancewic

University of Pennsylvania
stevez@cis.upenn.edu

Abstract

The C programming language is at least as well known for its absence of spatial memory safety guarantees (*i.e.*, lack of bounds checking) as it is for its high performance. C's unchecked pointer arithmetic and array indexing allow simple programming mistakes to lead to erroneous executions, silent data corruption, and security vulnerabilities. Many prior proposals have tackled enforcing spatial safety in C programs by checking pointer and array accesses. However, existing software-only proposals have significant drawbacks that may prevent wide adoption, including: unacceptably high runtime overheads, lack of completeness, incompatible pointer representations, or need for non-trivial changes to existing C source code and compiler infrastructure.



Defense-2: Shadow Stack

Shadow Stack

Traditional shadow stack

%gs:108

0xBEEF0048

Return address, R0
Return address, R1
Return address, R2
Return address, R3

Main stack

0x8000000

Parameters for R1
Return address, R0
First caller's EBP
Parameters for R2
Return address, R1
EBP value for R1
Local variables
Parameters for R3
Return address, R2
EBP value for R2
Local variables
Return address, R3
EBP value for R3
Local variables

Parallel shadow stack

0x9000000

Return address, R0

Return address, R1

Return address, R2

Return address, R3

Traditional Shadow Stack

```
SUB $4, %gs:108    # Decrement SSP
MOV %gs:108, %eax   # Copy SSP into EAX
MOV (%esp), %ecx    # Copy ret. address into
MOV %ecx, (%eax)    #      shadow stack via ECX
```

Figure 2: Prologue for traditional shadow stack.

```
MOV %gs:108, %ecx   # Copy SSP into ECX
ADD $4, %gs:108     # Increment SSP
MOV (%ecx), %edx    # Copy ret. address from
MOV %edx, (%esp)    #      shadow stack via EDX
RET
```

Figure 3: Epilogue for traditional shadow stack (overwriting).

Traditional Shadow Stack

```
MOV %gs:108, %ecx
ADD $4, %gs:108
MOV (%ecx), %edx
CMP %edx, (%esp) # Instead of overwriting,
JNZ abort        # we compare
RET
abort:
    HLT
```

Figure 4: Epilogue for traditional shadow stack (checking).

Overhead - Traditional Shadow Stack

If no attack:

- 6 more instructions

- 2 memory moves

- 1 memory compare

- 1 conditional jmp

Per function

Shadow Stack

Traditional shadow stack

%gs:108

0xBEEF0048

Return address, R0
Return address, R1
Return address, R2
Return address, R3

Main stack

0x8000000

Parameters for R1
Return address, R0
First caller's EBP
Parameters for R2
Return address, R1
EBP value for R1
Local variables
Parameters for R3
Return address, R2
EBP value for R2
Local variables
Return address, R3
EBP value for R3
Local variables

Parallel shadow stack

0x9000000

Return address, R0

Return address, R1

Return address, R2

Return address, R3

Parallel Shadow Stack

```
POP 999996(%esp) # Copy ret addr to shadow stack  
SUB $4, %esp # Fix up stack pointer (undo POP)
```

Figure 7: Prologue for parallel shadow stack.

```
ADD $4, %esp # Fix up stack pointer  
PUSH 999996(%esp) # Copy from shadow stack
```

Figure 8: Epilogue for parallel shadow stack.

Overhead Comparison

The overhead is roughly 10% for a traditional shadow stack.

The parallel shadow stack overhead is 3.5%.



Defense-3:

Stack cookies; Canary

specific to sequential stack overflow

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

Abstract:

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attacks gained notoriety in 1988 as part of the Morris Worm incident on the Internet. While it is fairly simple to fix individual buffer overflow vulnerabilities, buffer overflow attacks continue to this day. Hundreds of attacks have been discovered, and while most of the obvious vulnerabilities have now been patched, more sophisticated buffer overflow attacks continue to emerge.

We describe StackGuard: a simple compiler technique that virtually eliminates buffer overflow vulnerabilities with only modest performance penalties. Privileged programs that are recompiled with the StackGuard compiler extension no longer yield control to the attacker, but rather enter a fail-safe state. These programs require *no* source code changes at all, and are binary-compatible with existing operating systems and libraries. We describe the compiler technique (a simple patch to gcc), as well as a set of variations on the technique that trade-off between penetration resistance and performance. We present experimental results of both the penetration resistance and the performance impact of this technique.

StackGuard

A compiler technique that attempts to eliminate buffer overflow vulnerabilities

- No source code changes
- Patch for the function prologue and epilogue
 - Prologue: push an additional value into the stack (canary)
 - Epilogue: check the canary value hasn't changed. If changed, exit.

Buffer Overflow Example: code/overflowret4

```
int vulfoo()
{
    char buf[30];

    gets(buf);
    return 0;
}

int main(int argc, char *argv[])
{
    vulfoo();
    printf("I pity the fool!\n");
}
```

Use “echo 0 | sudo tee /proc/sys/kernel/randomize_va_space” on Ubuntu to disable ASLR temporarily

With and without Canary 32bit

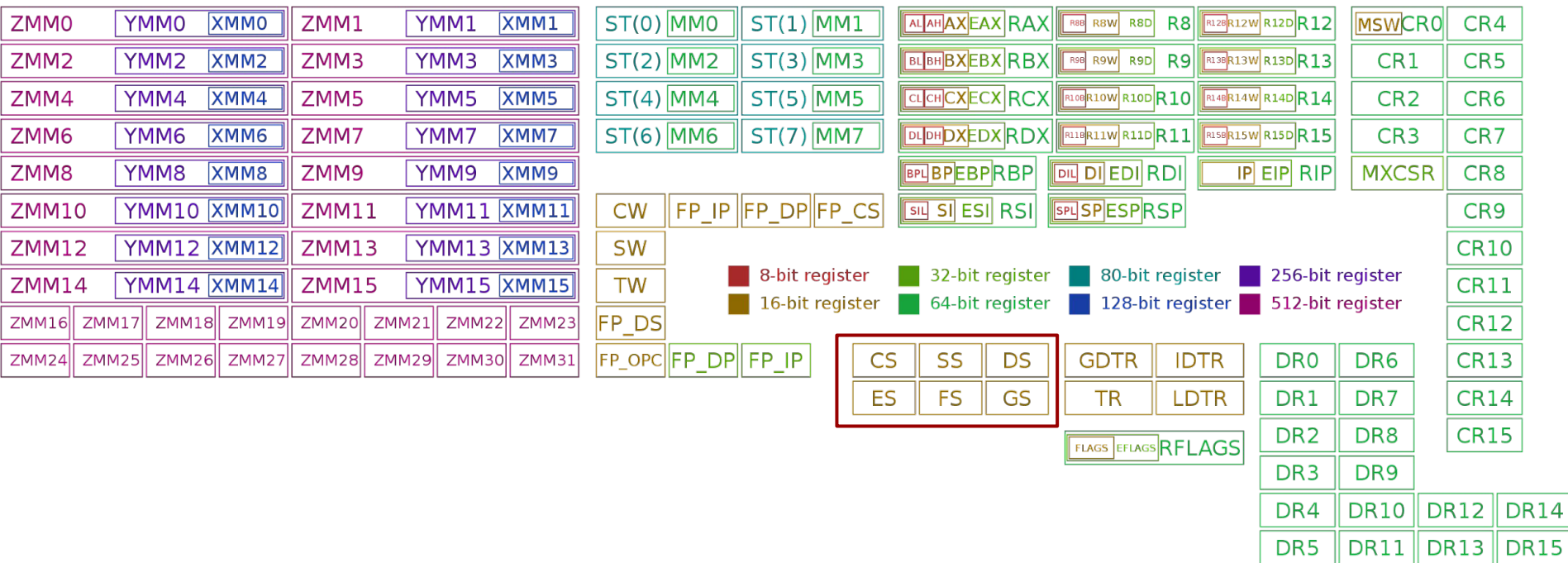
or4

```
000011ed <vulfoo>:
11ed:f3 0f 1e fb    endbr32
11f1: 55             push %ebp
11f2: 89 e5          mov %esp,%ebp
11f4: 53             push %ebx
11f5: 83 ec 34       sub $0x34,%esp
11f8: e8 64 00 00 00 call 1261 <_x86.get_pc_thunk.ax>
11fd: 05 d7 2d 00 00 add $0x2dd7,%eax
1202: 83 ec 0c       sub $0xc,%esp
1205: 8d 55 d0       lea -0x30(%ebp),%edx
1208: 52             push %edx
1209: 89 c3          mov %eax,%ebx
120b: e8 70 fe ff ff call 1080 <gets@plt>
1210: 83 c4 10       add $0x10,%esp
1213: b8 00 00 00 00 mov $0x0,%eax
1218: 8b 5d fc       mov -0x4(%ebp),%ebx
121b: c9             leave
121c: c3             ret
```

or4nx

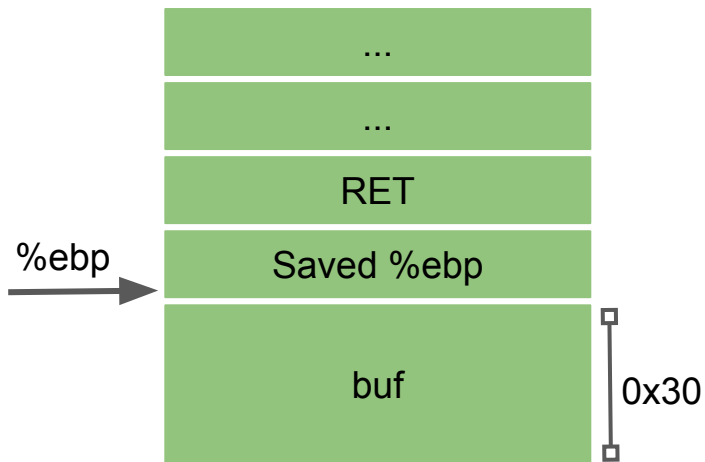
```
0000120d <vulfoo>:
120d:f3 0f 1e fb    endbr32
1211: 55             push %ebp
1212: 89 e5          mov %esp,%ebp
1214: 53             push %ebx
1215: 83 ec 34       sub $0x34,%esp
1218: e8 81 00 00 00 call 129e <_x86.get_pc_thunk.ax>
121d: 05 b3 2d 00 00 add $0x2db3,%eax
1222: 65 8b 0d 14 00 00 00 mov %gs:0x14,%ecx
1229: 89 4d f4       mov %ecx,-0xc(%ebp)
122c: 31 c9          xor %ecx,%ecx
122e: 83 ec 0c       sub $0xc,%esp
1231: 8d 55 cc       lea -0x34(%ebp),%edx
1234: 52             push %edx
1235: 89 c3          mov %eax,%ebx
1237: e8 54 fe ff ff call 1090 <gets@plt>
123c: 83 c4 10       add $0x10,%esp
123f: b8 00 00 00 00 mov $0x0,%eax
1244: 8b 4d f4       mov -0xc(%ebp),%ecx
1247: 65 33 0d 14 00 00 00 xor %gs:0x14,%ecx
124e: 74 05          je 1255 <vulfoo+0x48>
1250: e8 db 00 00 00 call 1330 <__stack_chk_fail_local>
1255: 8b 5d fc       mov -0x4(%ebp),%ebx
1258: c9             leave
1259: c3             ret
```

Registers on x86 and amd64

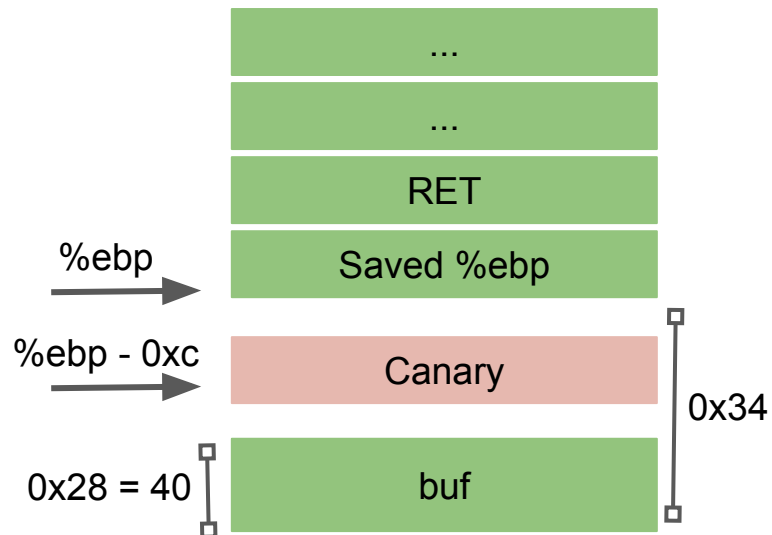


With and without Canary

or4



or4nx



With and without Canary 64bit

or464

```
0000000000001169 <vulfoo>:
 1169:f3 0f 1e fa    endbr64
 116d:55             push %rbp
 116e:48 89 e5        mov %rsp,%rbp
 1171:48 83 ec 30      sub $0x30,%rsp
 1175:48 8d 45 d0      lea -0x30(%rbp),%rax
 1179:48 89 c7        mov %rax,%rdi
 117c:b8 00 00 00 00   mov $0x0,%eax
 1181:e8 ea fe ff ff callq 1070 <gets@plt>
 1186:b8 00 00 00 00   mov $0x0,%eax
 118b:c9             leaveq
 118c:c3             retq
```

or464nx

```
0000000000001189 <vulfoo>:
 1189:f3 0f 1e fa    endbr64
 118d:55             push %rbp
 118e:48 89 e5        mov %rsp,%rbp
 1191:48 83 ec 30      sub $0x30,%rsp
 1195:64 48 8b 04 25 28 00 mov %fs:0x28,%rax
 119c:00 00
 119e:48 89 45 f8      mov %rax,-0x8(%rbp)
 11a2:31 c0           xor %eax,%eax
 11a4:48 8d 45 d0      lea -0x30(%rbp),%rax
 11a8:48 89 c7        mov %rax,%rdi
 11ab:b8 00 00 00 00   mov $0x0,%eax
 11b0:e8 db fe ff ff callq 1090 <gets@plt>
 11b5:b8 00 00 00 00   mov $0x0,%eax
 11ba:48 8b 55 f8      mov -0x8(%rbp),%rdx
 11be:64 48 33 14 25 28 00 xor %fs:0x28,%rdx
 11c5:00 00
 11c7:74 05           je 11ce <vulfoo+0x45>
 11c9:e8 b2 fe ff ff callq 1080 <_stack_chk_fail@plt>
 11ce:c9             leaveq
 11cf:c3             retq
```

Overhead - Canary

If no attack:

- 6 more instructions

- 2 memory moves

- 1 memory compare

- 1 conditional jmp

Per function

%gs:0x14, %fs:0x28

A random canary is generated at program initialization, and stored in a global variable (pointed by %gs, %fs).

Applications on x86-64 uses FS or GS to access per thread context including Thread Local Storage (TLS).

Thread-local storage (TLS) is a computer programming method that uses static or global memory local to a thread.

Pwngdb command *t/s* to get the address of tls

Data Structure

https://code.woboq.org/userspace/glibc/sysdeps/x86_64/nptl/tls.h.html

Canary Types

- Random Canary – The original concept for canary values took a pseudo random value generated when program is loaded
- Random XOR Canary – The random canary concept was extended in StackGuard version 2 to provide slightly more protection by performing a XOR operation on the random canary value with the stored control data.
- Null Canary – The canary value is set to 0x00000000 which is chosen based upon the fact that most string functions terminate on a null value and should not be able to overwrite the return address if the buffer must contain nulls before it can reach the saved address.
- Terminator Canary – The canary value is set to a combination of Null, CR, LF, and 0xFF. These values act as string terminators in most string functions, and accounts for functions which do not simply terminate on nulls such as gets().

Terminator Canary

0x000aff0d

\x00: terminates strcpy

\x0a: terminates gets (LF)

\xff: Form feed

\x0d: Carriage return

Evolution of Canary

StackGuard published at the 1998 USENIX Security. StackGuard was introduced as a set of patches to the GCC 2.7.

From 2001 to 2005, IBM developed ProPolice. It places buffers after local pointers in the stack frame. This helped avoid the corruption of pointers, preventing access to arbitrary memory locations.

In 2012, Google engineers implemented the `-fstack-protector-strong` flag to strike a better balance between security and performance. This flag protects more kinds of vulnerable functions than `-fstack-protector` does, but not every function, providing better performance than `-fstack-protector-all`. It is available in GCC since its version 4.9.

Most packages in Ubuntu are compiled with `-fstack-protector` since 6.10. Every Arch Linux package is compiled with `-fstack-protector` since 2011. All Arch Linux packages built since 4 May 2014 use `-fstack-protector-strong`.

ProPolice

```
int foo() {  
    int a;  
    int *b;  
    char c[10];  
    char d[3];  
  
    b = &a;  
    strcpy(c,get_c());  
    *b = 5;  
    strcpy(d,get_d());  
    return *b;  
}
```

Default Layout

RET
Saved %ebp
a
b
c
d

ProPolice

RET
Saved %ebp
Canary
c
d
a
b

Bypass Canary

-fstack-protector

Bypass Canary

1. Read the canary from the stack due to some information leakage vulnerabilities, e.g. format string
2. Brute force. 32-bit version. Least significant is 0, so there are 256^3 combinations = 16,777,216

If it take 1 second to guess once, it will take at most 194 days to guess the canary

Bypass Canary - Apps using fork()

1. Canary is generated when the process is created
2. A child process will not generate a new canary
3. So, we do not need to guess 3 bytes canary at the same time. Instead, we guess one byte a time. At most $256 * 3 = 768$ trials.

code/bypasscanary

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

char g_buffer[200] = {0};
int g_read = 0;

int vulfoo()
{
    char buf[40];
    FILE *fp;

    while (1)
    {
        fp = fopen("exploit", "r");
        if (fp)
            break;

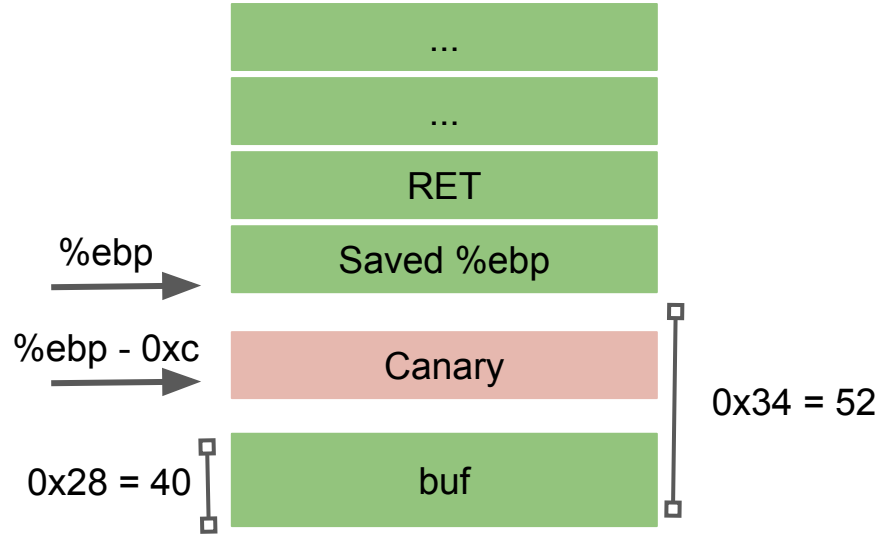
        usleep(500 * 1000);
        g_read = 0;
        memset(g_buffer, 0, 200);
        g_read = fread(g_buffer, 1, 70, fp);
        printf("Child reads %d bytes. Guessed canary is %x.\n",
            g_read, *((int*)&g_buffer[40]));
    }
}
```

```
        memcpy(buf, g_buffer, g_read);

        fclose(fp);
        remove("exploit");
        return 0;
    }

int main(int argc, char *argv[])
{
    while(1)
    {
        if (fork() == 0)
        {
            //child
            printf("Child pid: %d\n", getpid());
            vulfoo();
            printf("I pity the fool!\n");
            exit(0);
        }
        else
        {
            //parent
            int status;
            printf("Parent pid: %d\n", getpid());
            waitpid(-1, &status, 0);
        }
    }
}
```


bc



Canary: 0x??????00

Demo

1. Assume ASLR is disable.
2. To make things easier, we put the shellcode in env variable.
3. Write a script to guess the canary byte by byte.
4. Send the full exploit to the program

```
export SCODE=$(python -c "print '\x90'*500 +  
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b  
\xcd\x80\x31\xc0\x40xcd\x80'")
```