16:10 Locating Return Addresses via High Entropy Stack Canaries

by Matt Davis

Introduction

The following article describes a technique that can be used to identify a function return address within an opaque memory space. Stack canaries of maximum entropy can be used to locate stack information, thus repurposing a security mechanism as a tool for learning about the memory space. Of course, once a return address is located, it can be overwritten to allow for the execution of malicious code. This return address identification technique can be used to compromise the stack environment in a multi-threaded Linux environment. While the operating system and compiler are mere specificities, the logic discussed here can be considered for other executing environments. This all assumes that a process is allowed to inspect the memory of either itself or of another process.

Canaries and Stacks

Stack canaries are a mechanism for detecting a corrupted stack, specifically malware that relies on stack overflows to exploit a function's return address. Much like the oxygen-breathing avian in a coalmine, which acts as a primitive toxic-gas detector, the analogous stack canary is a digital species that will be destroyed upon stack corruption/compromise. Thus, a canary is a known value that is placed onto the stack prior to function execution. Upon function exit, that value is validated to ensure that it was not overwritten or corrupted during the execution of the function. If the canary is not the original value, then the validation routine can prematurely terminate the application, to protect the system from executing potential malware or operating on corrupted data.

As it turns out, for security purposes, it is ideal to have a canary that cannot be predicted beforehand. If such were not the case, then a crafty malware author could take control of the stack and patch the expected value over-top of where the canary lives. One solution to avoid this compromise is for the underlying system's random number generator (`/dev/urandom`) to be used for generating canary values. That is arguably a better solution to using hard-coded canaries; however, one can compromise a stack by using a randomly generated canary as a beacon for locating stack data, importantly return addresses. Before the technique is discussed, the idea of stacks living in dynamically allocated memory space must be visited.

POSIX threads and split-stack runtimes (think Go-lang) allocate threads and their corresponding stack regions dynamically, as a blob of memory marked as read/write. To understand why this is, one must first realize that threads are created at runtime, and thus it is undecidable for a compiler to know the number of threads a program might require.

Split-stacks are dynamically allocated thread-stacks. A split-stack is like a traditional POSIX thread stack, but instead of being a predetermined size, the stack is allowed to grow dynamically at runtime. Upon function entry, the thread will first determine if it has enough stack space to contain the stack contents of the to-be-executed function (prologue check). If the thread's stack space is not large enough, then a new stack is allocated, the function parameters are copied to the newly allocated space, and then the stack pointer register is updated to point to this new stack. These dynamically allocated stacks can still utilize the security implied by a stack canary. To illustrate the advantage of a split-stack, the default POSIX thread size on my box (created whenever a program calls '`pthread_create`') is hard-coded to 8MB. If for some reason a thread requires more than 8MB, the program can crash. As you can see, 8MB is a rather gross guess, and not quite scalable. With GCC's `-fsplit-stack` flag, threads can be created tiny and grow as necessary.

All this is to say that stack frames can live in a process' memory space. As I will demonstrate, locating stack data in this memory space can be simple. If a return address can be found, then it can be compromised. The memory mapped regions of thread memory are fairly easy to find, looking at '`/proc/<pid>/maps`' one can find the correspond memory maps. Those memory addresses can then be used to read or write to the actual memory located at '`/proc/<pid>/mem`'. Let's take a look at what happens after calling '`pthread_create`' once and dumping the maps table, as shown in Figure 4.

This figure highlights the regions of memory that were allocated for the threads, not all of this might be memory just for the thread. Note that the

```
00400000-00401000                r-xp 00000000 08:01 5505848  /home/user/a.out
00600000-00601000                r--p 00000000 08:01 5505848  /home/user/a.out
00601000-00602000                rw-p 00001000 08:01 5505848  /home/user/a.out
022c7000-022e8000                rw-p 00000000 00:00 0                [heap]
7fbdc8000000-7fbdc8021000        rw-p 00000000 00:00 0        <-- Thread memory.
7fbdc8021000-7fbdcc000000        ---p 00000000 00:00 0        <-- Guard memory.
7fbdcd18b000-7fbdcd18c000        ---p 00000000 00:00 0        <-- Guard memory.
7fbdcd18c000-7fbdcd98c000        rw-p 00000000 00:00 0        <-- Thread memory.
7fbdcd98c000-7fbdcdb27000        r-xp 00000000 08:01 7080135  /usr/lib/libc-2.25.so
[ ... Ignoring a few entries ... ]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0       [vsyscall]
```

Figure 4. Memory Map

pages marked without read and write permissions are guard pages. In the case of a read/write operation leaking onto those safety pages, a memory violation will occur and the process will be terminated.

This section started with an introduction with what a canary is, but what do they look like? The next two code dumps present a boring function and the corresponding assembly. This code was compiled using GCC's `-fstack-protector-all` flag. The `all` variant of this flag forces GCC to always generate a canary, even if the compiler can determine that one is not required.

The instruction '`movq %fs:40, %rax`' loads the canary value from the thread's thread local storage. This value is established at program load thanks to the libssp library (bundled with GCC). That value is then immediately pushed to the stack, 8 bytes from the stack's base pointer. The same compiler code that generated this stack push should also have generated the validation portion in the function's epilogue. Indeed, towards the end of the function there is a check of the stack value against the thread local storage value: '`xorq %fs:40, %rdx`.' If the values do not match, '`__stack_chk_fail`' is called to prematurely terminate the process.

```
// Boring function...
int foo(void){
    return 0xdeadbeef;
}

# In asm with -fstack-protector-all
# passed at compile time.
foo:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    %16, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    movl    $0xdeadbeef, %eax
    movq    -8(%rbp), %rdx
    xorq    %fs:40, %rdx
    je      .L3
    call    __stack_chk_fail
 .L3:
    leave
    ret
```

SUPERIOR MIGNON SYSTEM RADIO APPARATUS

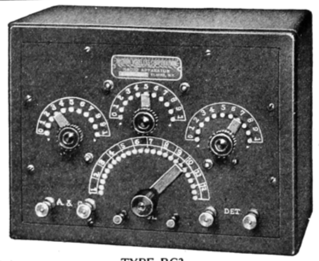
"Mignon System"

Apparatus of Scientific Construction for the Reduction of Static Interference

High Resonance—Unapproached Selectivity

NO TICKERS NOR ARMSTRONG CIRCUITS REQUIRED
for the reception of CONTINUOUS wave signals if you own a

MIGNON-SYSTEM CABINET



TYPE RC2

De Forest Audion Detectors and Amplifiers

BRANDES RECEIVERS

Crystaloi Detectors, Etc.

Write for R6 Catalog, Dept. "B"

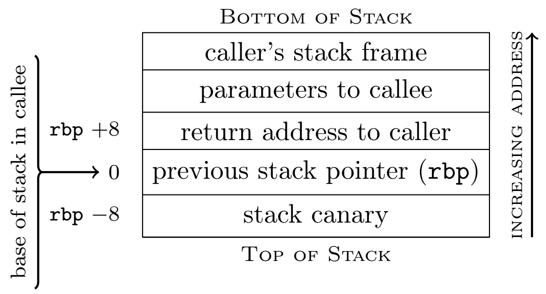
MIGNON WIRELESS CORPORATION

ELMIRA, N. Y., U. S. A.

Making use of Maximum Entropy to Identify a Stack

Now that we have gently strolled down thread-stack and canary alley, we now arrive at the intersection of pwnage. The question I am trying to answer here is: How can an malicious attacker locate a stack within a process' memory space and compromise a return address? I showed earlier what the `/proc` entry looks like, which can be trivial to locate by parsing the maps entries within the `/proc` file system. But how can one locate a stack within that potentially enormous memory space?

If your executable is at all security minded, it will probably be compiled with stack canaries. In fact, certain distributions alias GCC to use the `-fstack-protector` option. (See the man page of GCC for variations on that flag.) That is what we need, a canary that we can easily spot in a memory space. Since the canaries from GCC seem to be placed at a constant address from the stack base pointer, it also happens to be a constant address from the return address. The following is a stack frame with a canary on it. (This is x86, and of course the stack grows toward lower addresses.)



High entropy canaries simplify locating return addresses. Once a maximum entropy word has been located, an additional check can be made to see if the value 16 bytes from that word looks like an address. If that value is an address, it will fall within the bounds of any of the pages listed for that process in the `/proc` file system. While it is possible that it might be a value that looks like an address, it could also be a return address. At this point, you can patch that value with your bad wares.

The POC of this technique and the accompanying entropy calculation are included.[33] To calculate entropy I applied the Shannon Entropy formula, with the variant that I looked at bytes and not individual bits.

[33]`unzip pocorgtfo16.pdf canarypoc.c`

Afterward

As an aside, I scanned all of the processes on my Arch Linux box to get an idea of how common a maximum entropy word is. This is far from any kind of scientific or statistically significant result, but it provides an idea on the frequency of maximum entropy (bytes not bits). After scanning 784,700,416 words, I found that 4,337,624 words had a different value for each byte in the word. That is about 0.55% of the words being maximum entropy.

AVT OFERUJE:
LUTOWNICE Weller®
Groty proste/zgięte do serii SPI 14,90zł
▲ SPI-27C 230V 92,90zł
Subminiaturowa lutownica o mocy 25W, temp. grota 410°C
▲ SPI-16C 230V ...99,90zł
Subminiaturowa lutownica o mocy 15W temp. grota 360°C
▲ SPI-15 24V 89,90zł
STACJE LUTOWNICZE
◀ WTCP-S 464,90zł
Lutownica TCP-S, transformator 24V, podstawka KH-2.
WECP-20 .... 619,90 ▶
Lutownica 50W, transformator 24V, regulacja temperatury do 450°C, podstawka.
LUTOWNICE Elwik
STACJE LUTOWNICZE
LERT-24 .... 79,90zł ▲
Lutownica 60W, zasilana napięciem 24V. Wbudowany elektroniczny regulator temperatury. Zakres regulacji: 100°C...400°C.
▲ L–24–14 .... 24V/14W
L–24–18 .... 24V/18W
Lutownice o mocy 14 lub 18 W, bez regulacji temperatury, zasilane napięciem 24V. Temperatura grota: ok. 370°C.
W ofercie handlowej znajdują się także:
— odsysacze do lutowia z grzałką .... 49,90 zł
— tygielki elektryczne T-24 .... 47,00 zł
— groty do lutownic ELWIK .... 5,60 zł
▲ SEC–220–0 .... 294,90zł
Stacja lutownicza o mocy 60W Zakres regulacji: 100°C...400°C Cyfrowy odczyt temperatury grota.
Dostępne w sprzedaży wysyłkowej oraz w sklepach firmowych AVT
podane ceny nie zawierają podatku VAT (22%)