Post Memory Corruption Memory Analysis

Jonathan Brossard - jonathan.brossard@toucan-system.com

Security Research Engineer & CEO, Toucan System, France

Blackhat Briefings Conference, Las Vegas, 2011

"Normality is the route to nowhere." – Ridderstrale and Nordstorm

Abstract. In this article, we introduce a new exploitation methodology of invalid memory reads and writes, based on dataflow analysis after a memory corruption bug has occurred inside a running process.

We will expose a methodology which shall help writing a reliable exploit out of a PoC triggering an invalid memory write, in presence of modern security defense mechanism such as compiler enhancements (such as SSP...), libc protections (eg: safe heap unlinking), linking and dynamic linking enhancements (full read only GOT and relocations) or kernel anti exploitation features (ASLR, NX...).

In particular, we will demonstrate how to : find all the function pointers inside a running process, how to determine which ones would have been dereferenced after the Segmentation fault if the process had kept executing, which ones are truncatable (in particular with 0x00000000). In case all of the above fail, we will demonstrate how to test for overwrites in specific locations in order to indirectly trigger a second vulnerability allowing greater control and eventually full control flow hijacking. All of the above without needing the source code of the application debugged.

In the case of invalid memory reads, we will show how to indirectly influence the control flow of execution by reading arbitrary values, how to trace all the unaligned memory access and how to test if an invalid read can be turned into an invalid write or at least used to infer the mapping of the binary.

We will also introduce a new debugging technique which allows for very effective dynamic testing of all of the above by forcing the debugged process to fork(). All those steps are realized automatically and provide a rating of the best read/write location based on probabilities of mapping addresses (in the hope to defeat ASLR).

These techniques were implemented in the form of a proof of concept tool running under GNU/Linux and Intel architectures : pmcma[1].

Keywords: Exploit automation, post memory corruption analysis, debugging, memory protections, invalid memory writes.

[1] The official website of the tool is http://www.pmcma.org

Table of Contents

1 Introduction

Determining exploitability is hard, and writing exploits is hard. In fact, due to theoretical limitations (id est: "Halting Problem"), those two problems are the two sides of the same coin. Proving unexploitability is infeasible in the general case, and practically for the vast majority of computer programs actually used nowadays.

In this paper, we will examine exploitability in a systematic way, focusing on what happens in memory after a bug is triggered, rather than tracing or backtracking what has happened before. To the best of the author's knowledge, this is a new approach.

Our goal is to help exploit (semi)automation by building exploitation models based on constraints gathered from the environment (in particular, the presence of security countermeasures such as ALSR[1] or non-executable memory[2] thanks to kernel[3] or hardware[4] enhancements, as well as compiler enhancements such as Data Hardening[5], FORTIFY SOURCE[6] etc), and to allow for the practical testing of those models in order to (in)validate them.

We will primarily focus on invalid memory write bugs because of the special role they play in modern exploitation. Invalid memory dereferences in read mode used for the purposes of information leakage or indirect memory exploitation will also be discussed in this article.

The main contributions of this article are:
- A methodology to discover all the potential function pointers inside the address space of a process at any given point in time.
- A methodology to discover all the function pointers actually dereferenced by a process from a given point in time, given a fixed set of input data.
- A methodology to find all the function pointers exploitable by truncation in case of an arbitrary write subject to conditions (such as not controlling the value being dereferenced).
- A methodology to find all the unaligned memory reads from a given point in time during the execution of a process.
- A new debugging technique which allows the validation of all of the above, as well as the testing of arbitrary data modifications inside the address space of a process in order to achieve control flow hijacking from an arbitrary memory write.

All those techniques have been implemented in the form of a proof of concept tool for the GNU/Linux x86 Intel architectures. They could, without any loss of generality, be extended to any operating system or architecture with the exception of the last debugging technique, which requires the presence of the fork() system call and is therefore limited to *NIX operating systems[2].

This particular debugging technique doesn't require the debugged process to be restarted using execve(), and therefore preserves most of the mapping of the application (which may be hard to re-create because of the large entropy used in randomizing a process' address space under modern OSes). This technique is also believed to be the most effective to achieve this result (by construction) both in terms of speed and resources.

Finally, since overwriting function pointers doesn't allow direct shellcode execution anymore because of $W\hat{}X$ mappings, we introduce a new exploitation technique which works even under the most security enhanced kernels such as grsecurity. We call it "stack desynchronization". It allows frame faking inside the stack itself by having a controlled function pointer return to a carefully chosen function prologue instead of returning to a shellcode directly.
It is worth noting that we do not seek full exploit generation in this whitepaper, the output of our tool

[2] Actually, Security Researcher Mark Dowd made us aware that it may be possible to simulate a fork() under Windows too... Implementation details would be non trivial though, since the process creation mechanism is entirely different.

being a roadmap to exploitation rather than executable (or source) code. This roadmap needs to be implemented using both control flow and data flow analysis of the process prior to the bug, which is fortunately what virtually any existing debugging tool is capable of accomplishing.

2 Related work

Because of the theorical limitations invoked previously, earlier research on automatic exploitation tend to focus on numerical rather than analytical solutions. It is indeed less intellectually satisfying (it cannot be proved that a solution will actually be found in general), but interresting results have been achieved nonetheless.

More precisely, they all share a common underlying methodology : starting from a given program input leading to a deterministic memory corruption, they express the constraints on the input data in order to have it keep following the very same path (otherwise we're back to the Halting Problem and in practice to path exploration explosion), then turn each instruction into a set of constraints, express the desired result (id est: register set when receiving a Segmentation fault) in terms of the same constraints. Then they solve the equation numerically. Both taint analysis at assembly level (whether an intermediary language is used[7] or not[8]) and SAT Solvers[9][10] used on constraints expressed from C source seem to give satisfying results in practice, eventually modifying the input data to lead to the expected set of registers when triggering the bug, thereby leading to execution of arbitrary code.

Unfortunately, in practice the exploits created using the aforementioned techniques often do not work under realistic modern operating systems, in particular because the "complete exploits" generated automatically omit to take into account the additional complexity brought by security protections such as non-executable memory pages or Address Space Layer Randomization (ASLR).

If those techniques seem to give interesting results on simple vulnerabilities such as stack overflows under basic conditions (no stack canaries, no address space layer randomization and all sections being executable), they cannot cope with more complex vulnerabilities such as heap overflows (overwriting heap meta-data typically require having multiple memory locations set to appropriate values in order to pass the various libc checks before achieving a proper arbitrary write in memory) nor do they work with modern Operating Systems, which have enhanced kernels and compiler toolchains to prevent trivial exploitation.

In reality, dealing with those security protections is in itself what most security researchers and hackers alike actually spend their time on when writing exploits. It requires their exploitation methodologies to be environment-aware, those protections being implemented at different levels, ranging from compiler toolchains enhancements[6] to kernel modifications[11].

In the rest of this article, we will focus on what happens inside a process after it triggered a Segmentation Fault , assuming that the control flow and data flow analysis of input data leading to this Segmentation Fault, which are still the mandatory steps to determine exploitability, can be performed using commonly available tools (gdb, valgrind, dmalloc, electricfence,...).

3 mk_fork() : writing "weird debuggers" for "weird programs"

3.1 Motivation

An arbitrary anything/anywhere write allows an attacker to overwrite arbitrary data in sections mapped with writable permissions. In order to achieve control flow modification to execute arbitrary code, the most straight forward technique is to overwrite a function pointer that will later be dereferenced during the normal flow of execution.

Depending on the target binary, a few such pointers may be known without further reverse engineering. In fact, applications linked against the GNU libc library contain de facto such a pointer : the function pointer associated with the .dtors section. This pointer has long been used[12] to execute arbitrary code instead of the legitimate glibc destructors in case of arbitrary write vulnerabilities, particularly when exploiting missing format string vulnerabilities locally.

This particular technique, while popular in the early 2000's, has several limitations. First of all, it assumes the application will exit cleanly by calling exit(). If the attacker is unable to pursue normal execution much longer after triggering the vulnerability (fairly common in case of heap overflows, where heap metadata is irrecoverably corrupted and will sooner or later force the application to quit by calling abort() instead of exit(), hence not calling the normal destructors), the aforementioned function pointer will not even be dereferenced.

Secondly, achieving arbitrary remote execution when the application is exiting may be a bit late. If the attacker was to attack a remote service which typically only calls exit() when the server shuts down (which may literally take years), this attack vector may not be interesting at all. A carefully coded setuid application may also chose to drop privileges before exiting, hence wasting the attack vector for a local attacker.

Thirdly, guessing the location of the .dtors section, while easy if the binary wasn't compiled as a PIE is not immediately given in the opposite case, because the data section of the binary is then randomized. Finally, the application may not be linked with glibc at all (use of ulibc for instance), in which case, this function pointer is simply not available. In any case, it is possible to modify the linking process through a script[?] to make the .dtors section non writable, which mitigates this attack vector completely. This is acceptable for the vast majority of applications since the use of custom destructors is in fact not widespread.

The second popular technique is to overwrite a pointer in the Global Offset Table (GOT). When calling procedures whose code is stored in separate objects such as shared libraries, the lazy dynamic linking first transfers control to the Process Linkage Table (PLT), which is a trampoline to the GOT. Overwriting the GOT entry to say, printf() would allow an attacker to modify the flow of execution to an arbitrary location when printf() would later be called anywhere within the application. This technique is also attractive since binaries not compiled as Position Independent Executables (PIE) have their GOT stored at a fix location.

Unfortunately for the attacker, recent[5] modifications to the linker and dynamic linker allow[3] for the relocations to be performed entirely during the loading process. Since the GOT is then fully resolved, it can be set read only by a simple call to mprotect(). The net result is that an attacker cannot write to the GOT at all anymore, hence mitigating this attack vector entirely. This early binding comes at a performance cost (lazy binding no longer applies, and even references that would not have been used during a specific run of the application are resolved anyways), and is therefore not applied systematically to all binaries under all Linux distributions. That being said, it is absolutely possible to

[3] When compiling applications using gcc, and zith the following flags: "-Wl -z relro -z now". Using only "-Wl -z relro " allows for internal reorganization of the sections of the binary, putting the GOT before writable sections such as .data and .bss. This prevents GOT overwriting in case of user controlled buffer overflows in those writable sections. Adding the "-z now" option also forces relocations to be performed at load time, and enforce a call to mprotect() to render the GOT unwritable.

create a toolchain that would enforce those linking options to all the binaries in the system, effectively killing this attack vector entirely. The Gentoo Hardened distribution is such an instance of a distribution that privileged security over performance by enforcing this new feature by default on all binaries.

An other pointer prior researchers used in the past is the array of pointers called by at_exit(). While it has been shown[13] that overwriting a double word in this glibc global data could grant arbitrary code execution, applications actually using at_exit() tend to be fairly rare. The virtue of this example is generalize the overwrite of function pointers not only in the mapping of the application itself, but also in the writable mappings of the libraries it is linked to. In a way, the present paper can be seen as an extreme generalization of this technique.

If deferencing a function pointer is indeed a good idea, publicly available debuggers do a poor job at listing them. This is understandable since what happens inside an application after a Segmentation Fault is of little interest to normal software developers, whose focus is to fix bugs, not to write exploits. We will therefore without further due introduce a technique to automatically detect all the function pointers possibly dereferenced by an application (including its libraries) after a given memory corruption bug has occurred.

3.2 Methodology

Our methodology is based on the use of ptrace() to debug a process. We start by either attaching to a running process thanks to its pid, or create a new process from the command line of pmcma.

We then wait for the process to crash, emitting a Segmentation Fault caught by ptrace(). This indeed assumes that the user of the pmcma is able to create an input to create an invalid memory access inside the process.

On any given process, the amount of mapped memory is limited by architecture constraints. In order to verify if overwriting[4] a given double word in memory will in fact modify the flow of execution, we start by listing all the memory locations that are writable (we cannot modify non writable locations anyway). This preliminary phase is performed on a memory snapshot, but performing it after a Segmentation Fault allows us to have a binary which looks (in memory) exactly like the process we'd like to exploit from a mapping point of view.

We then force the debugged application to fork(). This creates a new process which only differs from the original debugged process by its process id. In particular, all the writable memory locations including the heap, or even global data from all the mapped libraries remain exactly the same.

We then overwrite a given writable location with a dummy value corresponding to a location non-executable in userland inside the newly created process (therefore leaving the original process intact, for later use). 0xf1f2f3f4 is a good such value[5]. We then clear the signals received by the newly spawned process and follow its execution as if a Segmentation Fault didn't just happen.

In case execution is transfered to memory location 0xf1f2f3f4 before the application exits, we have found a proper function pointer actually dereferenced by the application. This is easily detectable as it will trigger a SIGSEGV signal (the Segmentation Fault being due to an attempt to execute code in a location normally reserved to ring 0).

If such is not the case, we repeat the process of forcing the original debugged process to fork() and overwrite an other memory location with the same dummy pointer.

[4] simulating the fact that the previous insctruction triggering the Segmentation Fault had led to an arbitrary memory write instead of simply triggering a SEGFAULT...

[5] It is both always pointing to kernel land, regardless of the kernel split in use, and easy to identify.

By iterating this way over all of the possible writable memory locations, we can find all the function pointers dereferenced by the application during its normal flow of execution.

3.3 mk_fork() implementation

Previous works[14][15] have shown it was possible to use ptrace to inject an arbitrary library inside the process' address space. We don't need that much, we'll just inject a small shellcode forcing the process to call fork, and start ptracing the child.

Let's see how this can be achieved (ignoring error handling here and unnecessary complexity for the sake of clarity):

```
/*
*
* force a process to fork()
*
* returns the pid of the offspring
*
*/
int mk_fork(pid_t pid){

    void *target_addr;
    struct  user_regs_struct regz;
    struct  user_regs_struct regs;
    struct  user_regs_struct regz_new;
    int status;
    siginfo_t si;
    struct w_to_x_ptr *tmp4;
    int newpid;
    int fork_ok=0,offspring_ok=0;

    /*
    * prepare child to perform a fork
    */

    // save registers
    ptrace(PTRACE_GETREGS, pid,NULL, &regz);

    memcpy(&regz_new,&regz,sizeof(regz));

    // backup content at addr
    getdata(pid, (int)target_addr, backup_buff, 200);

    // replace with fork_stub shellcode
    write_data(pid,(int)target_addr,fork_stub,10);

    // execute fork_stub
    regz_new.eip=(int)target_addr+2;
    ptrace(PTRACE_SETREGS, pid,NULL, &regz_new);

    /*
     * Continue ptracing untill we get both a
     * SIGTRAP (parent) or SIGSTOP (child)
```

```
     */
    fork_ok=0;
    offspring_ok=0;
    while((!fork_ok)&&(!offspring_ok)){
        memset(&si, 0, sizeof(siginfo_t));

        ptrace(PTRACE_GETREGS, pid,NULL, &regs);
        ptrace(PTRACE_SETREGS, pid,NULL, &regs);

        ptrace(PTRACE_CONT, pid, NULL, NULL);
        waitpid(-1,&status,P_ALL); // any pid
        ptrace(PTRACE_GETSIGINFO, pid, NULL, &si);

        // parent ?
        if(si.si_signo == 5){
            fork_ok=1;
        }
        // offspring ?
        if (status >> 16 == PTRACE_EVENT_FORK) {
            ptrace(PTRACE_GETEVENTMSG, pid, NULL, (void*) &newpid);
            ptrace(PTRACE_SYSCALL, newpid, NULL, NULL);
        }
    }

    /*
    * Clean up the mess
    */
    // clear signals
    memset(&si, 0, sizeof(siginfo_t));
    ptrace(PTRACE_SETSIGINFO, pid, NULL, &si);
    ptrace(PTRACE_SETSIGINFO, newpid, NULL, &si);

    // restore data
    write_data(pid,(int)target_addr,backup_buff,200);
    write_data(newpid,(int)target_addr,backup_buff,200);

    // restore registers
    ptrace(PTRACE_SETREGS, pid, NULL, &regz);
    ptrace(PTRACE_SETREGS, newpid, NULL, &regz);

    return newpid;
}
```

With fork_stub being a small shellcode[6] :

```
;forking shellcode:
00000000  6631C0            xor eax,eax
00000003  B002              mov al,0x2
00000005  CD80              int 0x80
```

followed by 4 bytes 0xcc which will trigger a signal 5 (SIGTRAP) when executed.

[6] The shellcodes in this paper will be given assuming a 32b intel architecture for illustrative purpose.

The operations performed are therefore the following : first of, the state of the registers of the debugged application are saved. Then, 10 bytes from the debugged process are backed up, starting from target_addr (which is the place we will use to write and execute our small shellcode. This location needs to be mapped in an executable location). Our small shellcode is then injected inside the running process. Registers are then modified in the debugged process so that the next instruction to be executed will be our shellcode, and control is passed to this application. Upon correct execution of this shellcode, we will receive two signals : a SIGTRAP emitted by the debugged process, and a SIGSTOP emitted by its newly created offspring. We then restore the 10 bytes backed up earlier in both processes and restore their registers to the their original state.

This way, we obtain an almost perfect replica of our original process to experiment with at will.

3.4 Limitations

.

The main limitation is that all of the Inter Process Communication (IPC) and file I/O can be assumed to be in unpredictable state in the context of the offspring. The return of any syscall is in fact unpredictable. This may cause differences in the execution of the original process and the offspring. In particular, this may cause the offspring to exit earlier than the original process would have because of IPC or syscall errors, leading to false negatives in our analysis.

Experimentally, this experimental technique works well enough to provide reachable function pointers called, even though it still misses many that would exist much later in the flow of execution.

The system calls could probably be recorded in the original process and faked in offsprings to remove those problems entirely, thanks fo the ptrace() method PTRACE_SYSCALL. And a further cost in performance. This idea is further described later in this whitepaper under section 6.5.

4 Here be dragons : zombie reaping

The previously described methodology to create processes is indeed powerful, but creating unexpected children to a process poses several problems if we intend to debug large applications such as network services or web browsers. In this later case, we will need to analyze megabytes of writable data, hence create millions of children. In order to scale under those proportions, dealing with the termination of the created offsprings is mandatory.

4.1 Dealing with SIGCHLD

Since the original application will be kept sleeping while we will create thousands if bot millions of children to test writes in different locations, it won't be able to wait() for the return signal (SIGCHLD) emitted by each child process created when exiting. If we don't solve this situation, all those unreceived signals will prevent the child processes from actually terminating, leaving them in a zombie state.

First of, this is a waste of memory and cpu cycles because the zombies still have an entry in say, task_struct in kernel land. Those processes will also uselessly keep a process id, which is a limited resource on a computer. Once all the available process ids will be attributed to children processes eventually ending in zombie states, we will not be able to create new ones at all.

The first strategy to avoid zombies is to explicitly have the original process ask not to be sent SIGCHLD signals when its offsprings exit. This is fortunately possible under GNU/Linux by using sigaction() to ignore SIGCHLD signals. The kernel will then not bother sending signals to our dormant process.

The C code to perform this operation is equivalent to:

```
struct sigaction sa = {.sa_handler = SIG_IGN};
sigaction(SIGCHLD, &sa, NULL);
```

This code needs to be called only once by our original process. To perform this operation, we use the same injection methodology as with the mk_fork() shellcode. Our position independent shellcode stub to perform this system call is the following:

```
; Sigaction shellcode: // Zombie reaper
; struct sigaction sa = {.sa_handler = SIG_IGN};
; sigaction(SIGCHLD, &sa, NULL);

_start:
        nop
        nop
        nop
        nop
        call fake
fake:
        pop ecx
        add ecx,0x18     ; delta to sigaction structure

        xor eax,eax
        mov al,0x43      ; sigaction
        mov ebx,0x11     ; SIGCHLD
        xor edx,edx      ; 0x00
        int 0x80

        db 0xcc, 0xcc,0xcc,0xcc

; struct sigaction sa = {.sa_handler = SIG_IGN};
        db 01, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00
        db 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00
        db 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00
        db 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00
        db 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00
        db 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00
        db 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00
        db 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00
        db 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00
        db 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00
```

Using this technique, we don't need to care about pending signals anymore. But since our weird debugger is spawning so many children, we need to ensure that those processes actually terminate after a given period of time. And that any processes that they'd have spawned themselves without our knowledge will also terminate in order to spare computer resources.

4.2 Processes grouping

POSIX offers a great and little known way to solve this problem. Instead of systracing every created children and hook grandchildren creation, we can create process groups. Those groups are created for instance using a call to the syscall setpgid() to create a new group. All the offsprings of the process will then belong to this same group. Instead of killing processes one by one, we then kill the whole group using kill(-groupnumber).

There is a non POSIX but very efficient version of this syscall under Linux[7]. The prototype of this function is:

```
int setpgid(pid_t pid, pid_t pgid);
```

The description of this function, taken from the Linux man page gives:

```
setpgid()  sets  the  PGID of the process specified by pid to pgid.  If
pid is zero, then the process ID of the calling process  is  used.   If
pgid is zero, then the PGID of the process specified by pid is made the
same as its process ID.  If setpgid() is used to move  a  process  from
one  process  group to another (as is done by some shells when creating
pipelines), both process groups must be part of the same  session  (see
setsid(2)  and  credentials(7)).   In  this case, the pgid specifies an
existing process group to be joined and the session ID  of  that  group
must match the session ID of the joining process.
```

By injecting the following setpgid_stub inside a process, we can force it to create a new group:

```
;
; setpgid(0,0); shellcode
;

_start:
        nop
        nop
        nop
        nop
        mov eax,0x39    ; setpgid
        xor ebx,ebx
        xor ecx,ecx
        int 0x80

        db 0xcc, 0xcc
```

When called with 0 as a "pgid" group parameter, the process of the calling process is used as a group id, which is pretty handy as it avoids us to keep track of pids to groups associations.

Using this technique in addition to the previous zombie reaping one, we manage to keep the number of running processes arbitrary low even when debugging large applications such as web browsers. The Opera web browser (which is closed source) was for instance debugged this way to analyze CVE-2011-1824[16].

[7] Confere "man 2 setpgid" for differences.

5 Exploiting invalid memory writes

In this chapter, we will describe how pmcma can be used to help exploit different subclasses of invalid memory writes. We will start with the study of fully controlled invalid memory writes, where an attacker controls both the destination where to write to, and the content being written fully. We will then envisage other classes of bugs, where the attacker has less degrees of liberty : the case of overflows in different writable sections, then the one where the attacker doesn't control the data being written, and the special sub case of aligned memory writes.

5.1 Finding all the function pointers dereferenced after an invalid write

Pmcma can be run in two fashions in order to perform an analysis. The first one is to attach to a running process by providing its pid at the command line. It is particularly suited when auditing processes like network daemons. The second one is by providing pmcma the path of an ELF binary and a command line arguments to provide it. In both cases, in its default mode, pmcma will then wait for a segmentation fault to start its analysis.

5.2 Overflows

Overflows can be seen as a subclass of arbitrary writes where the write operation is performed sequentially over a given number of bytes (the size of the overflow). As opposed to the previous subclass of bugs, the attacker doesn't get to chose where the overwrite is performed. They may though, be able to control the size of the overflow and the content being overwritten.

Assuming the attacker has control on both the length of the overflow and the data overwritten, limiting the scope of the previous audit to the one section being overwritten will find all the relevant function pointers potentially overwritten.

More exploitation strategies are mentioned later in this paper in case such a pointer could not be found.

5.3 Partial overwrites and pointers truncations

Another common case happens when an attacker controls fully the location of the write, but has no control over the values being written. The tactic then used in order to achieve control flow hijacking is to attempt to overwrite a function pointer only partially. This technique is referred in the literature as pointer truncation.

Depending on the (uncontrolled, hopefully repeatable) value of the data being written, an attacker can attempt to perform either a lower bytes overwrite or a upper bytes overwrite. The goal is that once modified, the new function pointer still points to a memory section mapped as executable.

In case the truncated function pointer points to the same section as the original one (typically several bytes before or after), this de facto implies that the section is both writable and executable[8]. Obtaining arbitrary code execution is then a matter of having an appropriate shellcode, possibly preceded by a nop sled, mapped at the destination address of the modified pointer.

In case the truncated pointer points to a different section, and particularly when it is modified to point to a section which is executable but not writable (such as the .text of a library or of the main binary itself), control will be transfered to a location which has very little chances to hold user controlled opcodes. In other words, the behavior of the binary from this point is totally not predictable. That being said, as a last resort strategy, it may be a good option to an attacker in the hope of triggering almost immediately an other invalid memory access (which is very likely), and that this second indirect

[8] Such mappings still do exist on actual distributions, but the more hardened ones, having a better kernel in terms of security, can prevent such mappings entirely.

vulnerability will give him more control over the data being possibly written (this is not granted and is largely unpredictable). For this to happen in a deterministic fashion though, the return address has to remain unchanged between different mappings due to ASLR. This is quite possible in theory, for instance by returning to a fix .text location in a non PIE binary.

It is worth noting that being able to write only the value 0x00000000 is a very common such scenario. In particular, integer overflows occurring in iteration counters of copy loops typically allow an attacker to write passed the intended limits of a write, possibly to arbitrary locations. But the data being written then often comes from section paddings because a given copy is taking bytes passed the last mapped buffer of a section to re-write them at another location.

Pmcma allows for automatic testing of the above sub-cases, taking into account both lower bytes and higher bytes truncations. After collecting informations over the mapping of each section of the binary, including its permissions, and listing the existing function pointers, it is able to determine which ones are truncable to point to mapped memory, even across sections :

```
--[ Validating function pointers (relaxed mode):

 <*> Dereferenced function ptr at 0xbfb7ef4c (full control flow hijack)
     0xbfb7ef4c --> 0x080e5e58  // repeatability:0/100

 <*> Dereferenced function ptr at 0xbfb80fec (full control flow hijack)
     0xbfb80fec --> 0x080e5fa2  // repeatability:0/100

 <*> Dereferenced function ptr at 0xbfb8101c (full control flow hijack)
     0xbfb8101c --> 0x0804f94d  // repeatability:0/100
...

 <*> Dereferenced function ptr at 0xbfb7ef4c (full control flow hijack)
     0xbfb7ef4c --> 0x080e5e58  // repeatability:0/100

 <*> Dereferenced function ptr at 0xbfb80fec (full control flow hijack)
     0xbfb80fec --> 0x080e5ea2  // repeatability:0/100


 --> total : 186 validated function pointers
             (and found 8 additional control flow errors)

--[ Function pointers exploitable by truncation with 0x41424344:
At 0xb70ce070 : 0xb70c63c2 will become 0xb70c4142 (lower truncated by 16 bits, dest perms:RW)
At 0xb70e40a4 : 0xb70ca8f2 will become 0xb70c4142 (lower truncated by 16 bits, dest perms:RW)
At 0xb70ec080 : 0xb70e5e02 will become 0xb70e4142 (lower truncated by 16 bits, dest perms:RW)
At 0xb731a030 : 0xb7315da2 will become 0xb7314142 (lower truncated by 16 bits, dest perms:RW)
At 0xb73230a4 : 0xb732003a will become 0xb7324142 (lower truncated by 16 bits, dest perms:RW)
At 0xb732803c : 0xb7325a36 will become 0xb7324142 (lower truncated by 16 bits, dest perms:RW)
At 0xb76a80d8 : 0xb7325bf0 will become 0xb7324142 (lower truncated by 16 bits, dest perms:RW)
```

In the previous example taken from an analysis on the text editor nedit under Ubuntu 10.10, 186 function pointers actually dereferenced were found, after the starting point of the analysis. Assuming that the value being written is not controlled and is (0x41424344)[9], 7 of them can be truncated to point to valid memory. In this run, the destination permission was always reported as "RW", that is both

[9] This value is configurable from the command line, and pmcma can also use the value actually being written during the invalid memory access leading to the first segmentation fault - this is the default.

Readable and Writable. In case the current kernel allowed execution of sections mapped as writable but not explicitly flagged as writable (id est: the kernel doesn't support the NX feature and doesn't emulate for this application), having a shellcode stored at either 0xb70e4142 or 0xb7324142 will result in arbitrary code execution.
For the seek of completeness, here is an other example, performed this time with a truncation by 0x00000000 on 4b aligned addresses, on /bin/sudo:

```
--[ Function pointers possibly exploitable by 4 byte aligned truncation with 0x00000000:
At 0x08067135 : 0x4008039e will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08067b29 : 0x40080637 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08067b69 : 0x40080639 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08067d89 : 0x4007a933 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8a03d : 0x4007a7a2 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8a059 : 0x4007a7a5 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8a241 : 0x4007a7a0 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8a581 : 0x4007a7a0 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8b351 : 0x4000013e will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8b90a : 0x40004042 will become 0x40000000 (lower truncated by 16 bits, dest perms:RX)
At 0x08a8c361 : 0x4007a7c3 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8c761 : 0x4007a7c7 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8c861 : 0x4007a7c8 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8caa1 : 0x4007a7bd will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8d00a : 0x400007a8 will become 0x40000000 (lower truncated by 16 bits, dest perms:RX)
At 0x08a8d1b9 : 0x4007a7d1 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8d245 : 0x4000006f will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8d2a2 : 0x400a0001 will become 0x400a0000 (lower truncated by 16 bits, dest perms:RX)
At 0x08a8d2c5 : 0x4007a7d2 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a8d2cd : 0x404e1cd5 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a944e1 : 0x4007a952 will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a949e5 : 0x40403fbf will become 0x40000000 (lower truncated by 24 bits, dest perms:RX)
At 0x08a94b03 : 0x40449040 will become 0x40449000 (lower truncated by 8 bits, dest perms:RX)
At 0x08a94e02 : 0x4000404a will become 0x40000000 (lower truncated by 16 bits, dest perms:RX)
At 0x08a95067 : 0x40415c40 will become 0x40415c00 (lower truncated by 8 bits, dest perms:RX)
At 0x08a95077 : 0x40415040 will become 0x40415000 (lower truncated by 8 bits, dest perms:RX)
At 0x08a95087 : 0x40414640 will become 0x40414600 (lower truncated by 8 bits, dest perms:RX)
...
```

We can verify that this time, truncations of different sizes are possible, and that the destination address would be readable and executable. It corresponds to the .text of shared libraries, and returning there, while almost[10] guaranteed to lead to execution of executable code. But the result of returning savagely to an unexpected location is entirely non predictable and needs be tested, for instance using an other iteration of pmcma.

[10] It is in fact possible in theory to return to the middle of an opcode, hence corresponding to an invalid instruction

5.4 Discovering unaligned memory reads

Last but not least, an extreme sub-case of overwrites occurs when an attacker has not only no or little control over the data being written, but has additional constraints over where the write location is being performed. The author found himself in such a situation where only the value 0x00000000 could be written, and only on 4 byte aligned locations. This atypical case was due to a copy loop where the destination was user controlled because of an integer overflow, but where the destination would always be a multiple of 4 (many such memory initialization or copy loops process 4byte aligned memory zones, because compilers try to keep the data aligned in order to maximize cpu efficiency, and because the size of an atomic copy is also a multiple of 4).

Most function pointers inside an application are aligned on 4 byte boundaries. In this case, influencing the flow of execution by truncating a function pointer like previously is not possible, due to the additional constraints on the destination. In case we found an unaligned function pointer, this could be practical. Therefore, whenever an unaligned function pointer is found, pmcma flags it as remarkable (but this is honestly quite rare on Intel architectures).

In case no such unaligned function pointers could be found, those highly challenging memory corruption bugs would probably be regarded as never exploitable by most exploit writters. In fact, we believe they may still be used to influence the control flow of execution, but indirectly.

Since an attacker can under those conditions only alter data 4b aligned to be 0x00000000, which has little chances of being interesting, the idea is to manage to overwrite partially a given double word in memory that will later be read using an unaligned read by the cpu. Such unaligned reads are fairly rare, but can be listed using a unique technique we developped for pmcma.

The following code performs those operations:

```
int monitor_unaligned(int pid){
        struct  user_regs_struct regz;

keepexec:
        // Set align flag
        ptrace(PTRACE_GETREGS, pid,NULL, &regz);
        regz.eflags |=0x40000;
        ptrace(PTRACE_SETREGS, pid,NULL, &regz);

        while(1){
                siginfo_t si;
                memset(&si, 0, sizeof(siginfo_t));

                // continue tracing
                ptrace(PTRACE_CONT, pid, NULL, NULL);
                wait(NULL);

                // display received signals
                ptrace(PTRACE_GETSIGINFO, pid, NULL, &si);
                last_signal=si.si_signo;

                siginfo_t si;
                memset(&si, 0, sizeof(siginfo_t));

                // void error
                memset(&si, 0, sizeof(siginfo_t));
                ptrace(PTRACE_SETSIGINFO, pid, NULL, &si);
```

```
                // disassemble at current eip
                char raw[40];
                memset(raw,0x00,40);

                getdata(pid, regz.eip, raw, 40);

                char line[400];
                x86_insn_t insn;/* instruction */

                memset(line,0x00,400);

                x86_disasm((unsigned char*)raw, 40, 40, 0x00, &insn );
                x86_format_insn(&insn, line, sizeof line,intel_syntax);

                if(strlen(line)>1){
                        printf("%08X: %s\n",(unsigned int)regz.eip,line);
                }

                // display registers
                display_regs(line,regz);

                // set eip to next instruction
                ptrace(PTRACE_GETREGS, pid,NULL, &regz);
                regz.eip+=insn.size;
                ptrace(PTRACE_SETREGS, pid,NULL, &regz);

                // void error
                memset(&si, 0, sizeof(siginfo_t));
                ptrace(PTRACE_SETSIGINFO, pid, NULL, &si);

                goto keepexec;
        }

        return 0;
}
```

The idea is to set the unaligned flag in the EFLAGS register so that any subsequent unaligned memory access triggers a signal 7 (Bus Error), as per the intel manuals[17]. By then disassembing the latest instruction executed, parsing it to retrieve the registers used and performing a call to ptrace() using the PTRACE_GETREGS request, pmcma is able to retrieve the address of all unaligned reads and writes.

The following example shows how determining all the unaligned memory read and write access could be performed against the OpenSSH daemon running on a Fedora 15 computer.

```
[root@fedora-box pmcma]# netstat -atnp|grep ssh
tcp        0      0 0.0.0.0:22      0.0.0.0:*       LISTEN      7619/sshd
tcp        0      0 :::22           :::*            LISTEN      7619/sshd
[root@fedora-box pmcma]#
```

In a second terminal, the auditor initiates a ssh connection :

```
[endrazine@fedora-box ~]$ ssh localhost
```

On the first terminal are then listed all the unaligned memory accesses, along with the relevant information regarding the instruction executed and the value of registers during each access:

```
signo: 7 errno: 0 code: 1
00BD9FDF: mov   [edx-0x4], ecx
 ecx= 00000000
 edx= 214e57b6
signo: 7 errno: 0 code: 1
00BDA336: mov   ecx, [eax+0x6]
 eax= bfb3cb08
 ecx= 0000000a
signo: 7 errno: 0 code: 1
00BDA339: mov   [edx+0x6], ecx
 ecx= cae03591
 edx= 214e20cc
signo: 7 errno: 0 code: 1
00BDA33C: mov   ecx, [eax+0x2]
 eax= bfb3cb08
 ecx= cae03591
signo: 7 errno: 0 code: 1
00BDA33F: mov   [edx+0x2], ecx
 ecx= 60000000
 edx= 214e20cc
signo: 7 errno: 0 code: 1
00BDA336: mov   ecx, [eax+0x6]
 eax= 002beb49
 ecx= 0000000a
signo: 7 errno: 0 code: 1
00BDA33C: mov   ecx, [eax+0x2]
 eax= 002beb49
 ecx= b09f2035
signo: 7 errno: 0 code: 1
00BDA342: movzx ecx, [eax]
 eax= 002beb49
 ecx= 4a33dae7
signo: 7 errno: 0 code: 1
00BD5A55: mov   [edx-0x3], ax
 edx= 214e20e4
signo: 7 errno: 0 code: 1
00BDA339: mov   [edx+0x6], ecx
 ecx= 00c34ff4
 edx= 214e20c8
signo: 7 errno: 0 code: 1
00BDA33F: mov   [edx+0x2], ecx
 ecx= 00002d58
 edx= 214e20c8
signo: 7 errno: 0 code: 1
00BDA336: mov   ecx, [eax+0x6]
 eax= 002beb53
```

```
 ecx= 0000000a
signo: 7 errno: 0 code: 1
00BDA33C: mov   ecx, [eax+0x2]
 eax= 002beb53
 ecx= 382bc34a
signo: 7 errno: 0 code: 1
00BDA342: movzx ecx, [eax]
 eax= 002beb53
 ecx= 5b802b3e
signo: 7 errno: 0 code: 1
00BD5A52: mov   [edx-0x7], eax
 eax= 00000000
 edx= 214e20e4
signo: 7 errno: 0 code: 1
00BD5A55: mov   [edx-0x3], ax
 edx= 214e20e4
signo: 7 errno: 0 code: 1
00BD5A52: mov   [edx-0x7], eax
 eax= 00000000
 edx= 214e20e4
signo: 7 errno: 0 code: 1
00BD5A55: mov   [edx-0x3], ax
 edx= 214e20e4
```

Even if such unaligned memory accesses are rare, writing 0x00000000 to partially modify a 32 bits value that will then be read using an unaligned read can trigger secondary bugs inside the applications, possibly giving more control over the registers used in this second operation to an attacker.

6 ASLR and its limits

ASLR is a pretty effective way to prevent exploitation, based on statistics. If an attacker can make only a single try, for instance when exploiting a client side vulnerability, and if ASLR is fully enforced, then it may act as a very effective mitigation. The publicly available debugging tools usually lack section based ASLR testing,and when they have this feature, such as paxtest, they lack the capability to test the ASLR of a given binary in its entirety. In this chapter, we indent to outline a few limits of ASLR as well as describe how ASLR testing has been implemented in pmcma.

6.1 Effective testing of ASLR

As the astute reader may have noticed from previous examples, when reporting a finding, pmcma systematically appends a metric of repeatability, such as:

```
 <*> Dereferenced function ptr at 0xbfb7ef4c (full control flow hijack)
     0xbfb7ef4c --> 0x080e5e58  // repeatability:0/100
...
 <*> Dereferenced function ptr at 0xb76fc4bc (full control flow hijack)
     0xb76fc4bc --> 0xb76c3e20  // repeatability:0/100
```

This metric actually reflects the probability of a given mapping to reoccur at the very same location.

In order to compute those probabilities, pmcma starts by relaunching the target binary a great number of times (100 by default). For each execution, it records the base address of the mapping of each section. The metric displayed along with findings is then the highest probability to find a given section at a particular address:

```
--[ Performing ASLR tests:
[section:001]    /bin/su
  most probable address:0x08048000, proba=100/100
[section:002]    /bin/su
  most probable address:0x0804f000, proba=100/100
[section:003]    /bin/su
  most probable address:0x08050000, proba=100/100
[section:004]
  most probable address:0x08051000, proba=100/100
[section:005]    [heap]
  most probable address:0x0805e000, proba<001/100
[section:006]    /lib/ld-2.12.1.so
  most probable address:0xb7583000, proba<002/100
[section:007]    /lib/ld-2.12.1.so
  most probable address:0xb7584000, proba<002/100
[section:008]    /lib/ld-2.12.1.so
  most probable address:0xb758d000, proba<002/100
[section:009]    [vdso]
  most probable address:0xb758e000, proba<002/100
[section:010]
  most probable address:0xb758f000, proba<002/100
[section:011]    /usr/lib/locale/locale-archive
  most probable address:0xb75b6000, proba<002/100
[section:012]    /lib/libnss_compat-2.12.1.so
  most probable address:0xb75b8000, proba<002/100
[section:013]    /lib/libnss_compat-2.12.1.so
  most probable address:0xb75b9000, proba<002/100
[section:014]    /lib/libnss_compat-2.12.1.so
  most probable address:0xb75ba000, proba<002/100
[section:015]    /lib/libnsl-2.12.1.so
  most probable address:0xb7711000, proba<002/100
[section:016]    /lib/libnsl-2.12.1.so
  most probable address:0xb7713000, proba<002/100
[section:017]    /lib/libnsl-2.12.1.so
  most probable address:0xb7714000, proba<002/100
[section:018]
  most probable address:0xb7718000, proba<002/100
[section:019]    /lib/security/pam_rootok.so
  most probable address:0xb771a000, proba<002/100
[section:020]    /lib/security/pam_rootok.so
  most probable address:0xb771b000, proba<002/100
[section:021]    /lib/security/pam_rootok.so
  most probable address:0xb771c000, proba<002/100
[section:022]    /lib/libpam.so.0.82.2
  most probable address:0xb7727000, proba<002/100
[section:023]    /lib/libpam.so.0.82.2
  most probable address:0xb7728000, proba<002/100
[section:024]    /lib/libpam.so.0.82.2
  most probable address:0xb774f000, proba<002/100
[section:025]    /lib/libpam_misc.so.0.82.0
  most probable address:0xb7751000, proba<002/100
[section:026]    /lib/libpam_misc.so.0.82.0
  most probable address:0xb7752000, proba<002/100
[section:027]    /lib/libpam_misc.so.0.82.0
  most probable address:0xb776e000, proba<002/100
[section:028]
```

```
  most probable address:0xb776f000, proba<002/100
[section:029]    /lib/libc-2.12.1.so
  most probable address:0xbf7e4000, proba<002/100
```

Computing those probabilities in an effective way is in itself quite a challenge : when should one stop the process and assume it is fully mapped ? The technique should also cope with network daemon that bind ports and may pose an additional problem. If we execute say OpenSSH and wait for it to be fully loaded, record its mapping and shut it down, the port 22 will not be available immediately for rebinding.

To cope with those problems, pmcma attempts to record mappings right after the proper loading of the main binary and its associated shared libraries. To achieve this aim, pmcma runs the application while debugging it using the PTRACE_SYSCALL request of ptrace(). This allows pmcma to be made aware of any system call performed by the debugged process. It then maintains a list of system calls used during the loading of an application:

```
int allowed_syscalls[]={3,5,6,11,33,45,91,125,192,197,243};
/*Those syscalls are used during execve() and loading :

        read                            3
        open                            5
        close                           6
        execve                          11
        access                          33
        brk                             45
        munmap                          91
        mprotect                        125
        mmap2                           192
        fstat64                         197
        set_thread_area                 243*/
```

Whenever a system call performed by the debugged process doesn't belong to this white list, pmcma assumes that execution has already been transfered to the entry point of the application, and that the loading has therefore entirely been done. It then records the base address of each section of the mapping and kills the debugged process.

The net benefit of this technique is to allow for a very effective recording of mappings. The only mappings pmcma would actually missed are those performed much later during execution such as pluggings or shared libraries mapped by the application itself.

Finally, that the same aim can be reached in a simpler way by putting a breakpoint on the entry point of the binary[11].

[11] Thanks to Ivanlef0u for this idea.

6.2 Non Position Independant Executables

As mentioned before in this article, binaries not explicitly compiled as position independent executables do not have their sections randomised (only their share libraries if any, their heap and stack are).

If Linux distributions biased towards security instead of performance such as Gentoo Hardened with grsecurity kernels enforce PIE compilation on every single binary of the system, this is hardly the case for the vast majority of the Linux distributions.

Mainstream distributions such as Fedora or Ubuntu only impose PIE compilation on a carefully chosen set of binaries. Typically only network deamons.

It means that even setuid binaries such as /bin/su or network clients such as web browsers are not compiled as position independent executables, and have therefore some sections not randomized. This may not seem too bad at first sight, but it really means that when looking for a fix pivoting address inside the application, an attacker is guaranteed to find some. This may be used not only to write 100% reliable ret2plt bootstrap shellcode in case of stack overflows, but also sometimes return to the .text of the binary as explained earlier in case of pointer truncations. We will see that this weakness can also be used to infere the mapping of the whole application when attempting to leak the layout of the binary towards the end of this chapter.

6.3 Prelinking

Prelinking is a time saving feature, employed notably by default on Fedora. It allows for faster loading of applications by precomputing the location of the shared libraries inside a process, and hardcoding those locations on disk.

The Fedora prelinking is renewed every two weeks thanks to a cron job. It means that during 14 days, the mapping of the shared libraries of a given process are entirely deterministic. Under pmcma, this means that the probability associated with the mappings of a given section fo a shared library will be of 100%.

Fedora's documentation explicitly mentioned this behavior[18] and concludes that the risk is acceptable since the mapping of shared library is chosen randomly every two weeks. In particular it will differ from machine to machine. We will see later in this chapter the limits of those assumptions : if an attacker could somehow retrieve the mapping of a given process at a given point in time, he would then know the mapping of subsequent executions of this same binary for some time.

6.4 Biased ASLR

Finally, it is worth mentioning that some distributions have very biased ASLR, due to improper kernels. This allows for probabilistic exploitation of binaries.
Here is an example of an analysis performed by pmcma on Ubuntu 10.10 with a kernel 2.6.32-26-generic:

```
--[ Performing ASLR tests:
[section:001]    /bin/ping
  most probable address:0x08048000, proba=100/100
[section:002]    /bin/ping
  most probable address:0x08050000, proba=100/100
[section:003]    /bin/ping
  most probable address:0x08051000, proba=100/100
[section:004]
  most probable address:0x08052000, proba=100/100
[section:005]    [heap]
  most probable address:0xb76a7000, proba<003/100
```

```
[section:006]    /lib/tls/i686/cmov/libnss_files-2.10.1.so
  most probable address:0xb76a9000, proba<003/100
[section:007]    /lib/tls/i686/cmov/libnss_files-2.10.1.so
  most probable address:0xb77e7000, proba<003/100
[section:008]    /lib/tls/i686/cmov/libnss_files-2.10.1.so
  most probable address:0xb77e8000, proba<003/100
[section:009]
  most probable address:0xb77ea000, proba<003/100
[section:010]    /lib/tls/i686/cmov/libc-2.10.1.so
  most probable address:0xb77eb000, proba<013/100
[section:011]    /lib/tls/i686/cmov/libc-2.10.1.so
  most probable address:0xb77ee000, proba<014/100
[section:012]    /lib/tls/i686/cmov/libc-2.10.1.so
  most probable address:0xb77fe000, proba<013/100
[section:013]    /lib/tls/i686/cmov/libc-2.10.1.so
  most probable address:0xb77ff000, proba<013/100
[section:014]
  most probable address:0xb7800000, proba<003/100
[section:015]    /lib/tls/i686/cmov/libresolv-2.10.1.so
  most probable address:0xb7810000, proba<003/100
[section:016]    /lib/tls/i686/cmov/libresolv-2.10.1.so
  most probable address:0xb7812000, proba<003/100
[section:017]    /lib/tls/i686/cmov/libresolv-2.10.1.so
  most probable address:0xb7813000, proba<003/100
[section:018]
  most probable address:0xb782e000, proba<003/100
[section:019]
  most probable address:0xb782f000, proba<003/100
[section:020]    [vdso]
  most probable address:0xbfa7d000, proba<002/100
```

In this analysis, pmcma was able to report that some shared libraries such as the libc actually have a given base address for their mapping much more probable (up to 13% of the time) than expected.[12]. If upgrading to a more recent kernel shipped by Ubuntu fixes this particular problem, it fundamentally means that custom kernels compiled by system administrators not sufficiently knowledgeable about security can lead to weak ASLR.

6.5 Memory mapping leakage

Previous researches[19][20], and in particular the 2010 WTFuzz exploit against IE8 under Windows 7[21] which won the pwn2own contest have shown that using JavaScript and a heap overflow to overwrite the NULL terminator of a Javascript string, it was possible for an attacker to be given more information than he should have accessed (when reading from the string in question). If the leaked bytes (which may be in random quantity, up to the next NULL byte) contained a pointer to data in other sections, then the attacker could infer the location of a mapping of a given section inside the running process (from JavaScript itself) and trigger a very precise second write to obtain arbitrary code execution.

To further generalise this technique, let's take a step back and look at the problem from a kernel's stand point. Essentially, all the information sent to an attacker use only a few system calls. Namely sys_write() and sys_socketsyscall(). The later offers a few different requests and now handles what used to be all the other socket related system calls, such as sys_connect(), or sys_send(). Let's have a look at the code of this system call in kernel 2.6.39 source code[13]:

[12] The analysis also shows that the code, data and read only data segments of ping are not randomized at all, but this was actually expected given that this binary isn't compiled as PIE.

[13] Sample code taken from net/socket.c

```
2234 SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)
2235 {
2236         unsigned long a[6];
2237         unsigned long a0, a1;
2238         int err;
2239         unsigned int len;
2240
...
2247
2248         /* copy_from_user should be SMP safe. */
2249         if (copy_from_user(a, args, len))
2250                 return -EFAULT;
2251
2252         audit_socketcall(nargs[call] / sizeof(unsigned long), a);
2253
2254         a0 = a[0];
2255         a1 = a[1];
2256
2257         switch (call) {
2258         case SYS_SOCKET:
2259                 err = sys_socket(a0, a1, a[2]);
2260                 break;
2261         case SYS_BIND:
2262                 err = sys_bind(a0, (struct sockaddr __user *)a1, a[2]);
2263                 break;
2264         case SYS_CONNECT:
2265                 err = sys_connect(a0, (struct sockaddr __user *)a1, a[2]);
2266                 break;
2267         case SYS_LISTEN:
2268                 err = sys_listen(a0, a1);
2269                 break;
2270         case SYS_ACCEPT:
2271                 err = sys_accept4(a0, (struct sockaddr __user *)a1,
2272                                   (int __user *)a[2], 0);
...
2287         case SYS_SEND:
2288                 err = sys_send(a0, (void __user *)a1, a[2], a[3]);
2289                 break;
2290         case SYS_SENDTO:
2291                 err = sys_sendto(a0, (void __user *)a1, a[2], a[3],
2292                                  (struct sockaddr __user *)a[4], a[5]);
...
2327         default:
2328                 err = -EINVAL;
2329                 break;
2330         }
2331         return err;
2332 }
```

In a nutshell, to call sys_socketcall, eax has to worth 102, then ebx specifies which particular call is to be performed, according to the following requests, defined in include/linux/net.h:

```
 26 #define SYS_SOCKET      1               /* sys_socket(2)               */
 27 #define SYS_BIND        2               /* sys_bind(2)                 */
 28 #define SYS_CONNECT     3               /* sys_connect(2)              */
 29 #define SYS_LISTEN      4               /* sys_listen(2)               */
 30 #define SYS_ACCEPT      5               /* sys_accept(2)               */
 31 #define SYS_GETSOCKNAME 6               /* sys_getsockname(2)          */
 32 #define SYS_GETPEERNAME 7               /* sys_getpeername(2)          */
 33 #define SYS_SOCKETPAIR  8               /* sys_socketpair(2)           */
 34 #define SYS_SEND        9               /* sys_send(2)                 */
 35 #define SYS_RECV        10              /* sys_recv(2)                 */
 36 #define SYS_SENDTO      11              /* sys_sendto(2)               */
 37 #define SYS_RECVFROM    12              /* sys_recvfrom(2)             */
 38 #define SYS_SHUTDOWN    13              /* sys_shutdown(2)             */
 39 #define SYS_SETSOCKOPT  14              /* sys_setsockopt(2)           */
 40 #define SYS_GETSOCKOPT  15              /* sys_getsockopt(2)           */
 41 #define SYS_SENDMSG     16              /* sys_sendmsg(2)              */
 42 #define SYS_RECVMSG     17              /* sys_recvmsg(2)              */
 43 #define SYS_ACCEPT4     18              /* sys_accept4(2)              */
 44 #define SYS_RECVMMSG    19              /* sys_recvmmsg(2)             */
```

So, to monitor all the data leaving the process and possibly reaching an attacker, being it over sockets, files, ttys or any other mean, all we need to pay attention to is sys_write() (syscall 4 under Intel x86 architectures), and sys_socketcall() (syscall 102 under Intel x86 architectures) for a few carefully chosen sub-calls: sys_send(), sys_sendto(), sys_sendmsg().

The main idea is to proceed as following: first of, make the original process fork() once. Unlike with previous techniques, we then let the original process run, and monitor it using the ptrace() PTRACE_SYSCALL request, which allows us to break every time the process will perform a system call. We record all the system calls executed, as well as their return data and values. We now have a reference run to compare subsequent experiments with.

Then, we make the saved offspring and make it fork(). We overwrite the first writable location in memory with dummy data. We then trace its execution thanks to the same ptrace() PTRACE_SYSCALL request. Everytime this processattempts to execute a system call, we compare it's input registers with the one of the original process. If the syscall to be executed is either sys_write(), or sys_socketcall() with a relevant sub-call, we verify if the data to be processed differs from the one of the original process.

Three cases may arise : the amount of data sent may differ. If it is now larger, we have found a location in memory, which, when overwritten, forces the application to send more data than expected back to the attacker. This would be the case when overwritting for instance a variable stored in a read/write section, and used as the length argument in the following statement:

```
write(3,&buff,length);
```

The second case happens when the data is entirely different, because for instance we would have overwritten a pointer to the buff variable in the previous case.
The third case is when both the data and the length differ entirely, for instance when overwritting a pointer to buff and then calling the pointer to data and then calling the following system call via sys_socketcall():

```
sendto(sockfd,&buff,sizeof(buff),0);
```

In all of those cases, we can actually with a pretty high accuracy verify if an interresting memory leak occured, which will allow an attacker to deduce the mappings of the binary. For this condition to occur, the leaked data (either new trailing bytes, or entirely different data sent back to the attacker) needs to contain a pointer to any section in the binary. Because of the way ASLR is performed under Linux (all the sections but heap and stack being translated by a constant offset), knowing a single pointer to the main binary or to a shared library will result in knowledge of the almost whole mapping. To discover the location of the heap or stack, we would need in addition to find a pointer to the heap (possible) and the stack (less realistic) in the data sent to the attacker. This is really only a matter of parsing the new data sent by the process, and match potential pointers against the memory addresses of each section in its address space.

We mentioned earlier that one of the biggest limitation of pmcma is the fact that system calls performed by offsprings could provide a different result than in the original process provided the same inputs (because sockets will now be closed, file descriptors in undefined states...). Since we have now described a method to record the system calls performed by the original process, it is possible to fake them in the offsprings (by using the ptrace() PTRACE_SYSCALL until a system call is to be called, and modify the ouput registers and optionally their associated data before adjusting eip : we don't even need to actually perform a real system call). The main inconvenient of this technique is the fact that some system calls pass data in non standard ways (eg: sys_socketcall()). We could extend pmcma to know how each syscall expects and modifies data, but there are about 300 of them in a modern Linux kernel, and they are architecture specific. Also, using PTRACE_SYSCALL has a non negligeable overhead in terms of performance.

7 Extending the capabilities of pmcma

We have so far focused on exploitation of invalid memory writes through the use of function pointer. Pmcma is capable of much more, and the capabilities offered in terms of exploitation modeling by the mk_fork() technique haven't been fully explored yet. In this chapter, we will describe a few distinctive features of pmcma.

7.1 Call tables and returns to registers+offsets

Pure function pointers are not the only way to directly modify the flow of execution of an application given an arbitary write bug. For exemple, redirection of the control flow via call tables and direct modifications of the control flow based on the value of a register, such as jmp [eax+0xdeadbeef] or call [ebx+0xc0f33babe] could be influenced in case an attacker could perform a controlled write when exploiting an invalid write vulnerability.

Pmcma is also able to detect the occurrence of such scenarios when attempting to write to different locations in the writable sections of an application. When reporting control flow modifications, it will differentiate the case where the value it has written to memory is the exact address later being dereferenced (labeled as "direct control flow hijack"). and the case where the it differs ("indirect control flow bug"):

```
...
 <*> Dereferenced function ptr at 0xb73ce08c (full control flow hijack)
     0xb73ce08c --> 0xb734e54e  // repeatability:2/100

 <*> Dereferenced function ptr at 0xb73de0a4 (full control flow hijack)
     0xb73de0a4 --> 0xb73d19aa  // repeatability:2/100

 <-> Triggered an indirect control flow bug when writing at 0xb73df000
     (ret value=0xf1f3c38c is unmapped)
     0xb73df000 --> 0xb73bf000  // repeatability:2/100

 <-> Triggered an indirect control flow bug when writing at 0xb73df2b0
     (ret value=0xf1f8ef7c is unmapped)
     0xb73df2b0 --> 0xb7348000  // repeatability:2/100
...
 --> total : 186 validated function pointers
            (and found 8 additional control flow errors)
```

In the previous example, the addresses where execution was attempted by the application (0xf1f3c38c and 0xf1f8ef7c) because of an indirect control flow bug are very close to the remarkable test value used (0xf1f2f3f4), which is a string indicator that the application in fact added an offset to this base value inside a register before attempting to jump (or call) the corresponding address.

7.2 Searching for pointers to structures (containing function pointers)

Since all sections do not always share the same amount of entropy, in particular when biased ASLR has been detected, it is tempting to attack the worst protected sections first. In case function pointers were found only in the best randomized sections, pmcma is able to perform yet an other analysis in order to maximize the effectiveness of exploitation.

Instead of attacking function pointers directly, it may be worth searching for pointers to data structures in other sections (the more heavily randomized ones) containing function pointers.

The attack scenario would then be the following : instead of overwriting a function pointer directly, overwrite the pointer (whose address is less randomized) to the structure to point to a user controled, writable location. Then fake the structure in this location, and eventually dereference the function pointer.

The conditions for this attack to work are quite realistic in many cases. For instance, data structures containing function pointers created by the application itself are typically stored on the heap, which is always heavily randomized, and not a good target for a blind overwrite in terms of probability. But if those function pointers are in fact stored in a linked list, that the first pointer of the liked list is stored on the proper data section of the application, and that the location of this first pointer can be guessed (for example because the binary isn't PIE), then overwriting the first pointer to point into a user controlled buffer in the data section itself would do the trick.

In order to detect pointers to structures containing function pointers, pmcma first parses the writable sections of the binary and search for possible pointers to other writable sections. Those will be the candidate pointers.

Then, it creates a new mapping inside offsprings created by mk_fork(). Those mapping should never be read or written to under normal conditions since they have been artificially created. Then pmcma modifies one pointer candidate per offspring to point to the beginning of the new mapping.

In case this modification triggers an invalid memory access in execution, pmcma deduces it has in fact overwriten a pointer to a structure containing a pointer actually dereferenced.

The created mappings also contain a particular pattern of bytes, which helps in identifying at which offset inside the mapping a function pointer is being dereferenced.

The algorithm to create a new mapping inside an offspring relies on the injection of a small stub shellcode to allocate memory via mmap(). The main idea of injecting a shellcode in a debugged process should be pretty familiar to the reader by now. The creation of the mapping then only requires to read the return address of mmap(), which is indeed stored into eax.
The shellcode used to achieve a proper memory allocation is given below:

```
;
; old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, 0, 0) shellcode:
;

_start:
        nop
        nop
        nop
        nop

        xor eax, eax
        xor ebx, ebx
        xor ecx, ecx
        xor edx, edx
        xor esi, esi
        xor edi, edi

        mov bx, 0x1000          ; 1 page
        mov cl, 0x3             ; PROT_READ|PROT_WRITE
        mov dl, 0x21            ; MAP_SHARED|MAP_ANON

        push eax
        push eax
```

```
        push edx
        push ecx
        push ebx
        push eax

        mov ebx, esp
        mov al, 0x5a            ; sys_mmap
        int 0x80

        ; eax contains address of new mapping

        db 0xcc, 0xcc, 0xcc, 0xcc
```

A typical analysis by pmcma would then look like:

```
--[ Searching pointers to datastructures with function pointers

 ** Pointers to +W sections: 15928

 <*> Dereferenced a function pointer inside a structure when writing at 0x094568e4
     (ret value=0xffffffff) // repeatability:1/100

 <*> Dereferenced a function pointer inside a structure when writing at 0x094616fc
     (ret value=0xffffffff) // repeatability:1/100

 <*> Dereferenced a function pointer inside a structure when writing at 0x094ac960
     (ret value=0xffffffff) // repeatability:1/100

 <*> Dereferenced a function pointer inside a structure when writing at 0x094aca10
     (ret value=0xffffffff) // repeatability:1/100

 <*> Dereferenced a function pointer inside a structure when writing at 0x094acab8
     (ret value=0xffffffff) // repeatability:1/100

 <*> Dereferenced a function pointer inside a structure when writing at 0xbfb81098
     (ret value=0xffffffff) // repeatability:0/100

 --> total : 6 function pointers identified inside structures
```

In this example, the return value is 0xffffffff, which corresponds to the padding of our newly created mapping. It is therefore not possible to immediately deduce from the mapping pattern at which offset inside this new mapping, the function pointer was located.

7.3 Testing exhaustively arbitrary writes

Since pmcma has the capabilities to make the debugged process fork() at will, it can exhaustively attempt to overwrite all the writable addresses mapped inside a given process, in the hope to trigger invalid memory access in execution mode. This process is not only slow and resource consuming, but pmcma cannot attempt to overwrite all those locations with all of the more than 4 billion possible values a 32b register allows. In consequence, this option is kept as a last resort in case all of the other strategies failed. It is nonetheless practical to overwrite all the possible locations inside a given process with a predefined remarkable value.

This feature may seem anecdotal at first sight. But it is currently the only way for pmcma to find the pointers associated with unresolved procedure relocations. The alternative would be to run all audits with an LD_BINDNOW environment variable set in order to force resolution by the dynamic linker at load time. Unfortunately, this isn't practical for analysis of network daemon, at least not without restarting them. In addition, the use of LD_BINDNOW would induce modifications inside the writable mappings of the binary and would no longer reflect its actual state in real exploitation conditions.

7.4 Testing invalid reads

Invalid reads by themselves do not allow direct modification of the control flow. They can nonetheless be interesting, depending on how this particular memory read is handled inside the application. If the value read by the faulting instruction is user controlled (meaning : the application can be forced to read from a given address in memory which is user controlled), it may trigger indirect invalid memory accesses either in execution or write modes. A trivial example would be an application using the value just read as a counter in ecx to perform a memory copy. By setting this register to a very large value, an attacker would indirectly cause an invalid memory access in this loop.

Testing for such indirect problems caused by an invalid read is fairly straight forward : by setting the register in which the value is read to multiple values in different offsprings of the debugged process, it is possible to detect if they would eventually result in an invalid memory access more interesting (either in write or execution mode) later one, simply by ptracing the offspring and disassembling the faulting address in case a Segmentation Fault was detected.

There again, testing the 2 ˆ32 possible values offered by modest 32b processors is most probably a bit overkill. Testing on a thousand of evenly spaced values across the search space is much more time saving and would spot most indirect vulnerabilities anyways.

8 Stack desynchronization

For the most part, we have focused so far on where to write in memory in order to achieve a modification of the control flow. It is about time we also consider the question of what to write. In other word, to consider what an hijacked function pointer should be modified to point to.

In case writable sections are found to be executable, and at least one is both reasonably controlled and not too randomized, the answer is quite simple : copying a nop sled and shellcode in at this position would grant an attacker arbitrary code execution. This is how exploitation has been achieve for about 15 years.

But this scenario is becoming less and less likely, in particular because writable sections are not executable anymore thanks either to PaX or cpu NX-like bits. In particular, the heap, which is the only section that can be made almost arbitrary big (which would help in making the exploit probabilistically better thanks to a classic heap spray) is not commonly executable anymore under GNU/Linux.

In order to overcome those problems, we suggest to return, not to a writable section, but to a carefully chosen function prologue. This indeed requires that such a prologue is either available in a non randomized section (.text of the binary if compiled without PIE[14]), or at an address we can predict (for instance thanks to a memory leak induced by a previous memory write, like explained earlier in this paper).

By returning to a chosen function prologue, an attacker will get to chose by how much he will modified the stack pointer. If they control a large buffer in the stack and can create fake stack frames in it, then he can default back to more standard stack-smashing-like exploitation (ret2libc, ret2plt or ROP depending on randomization and compilation options).

To the best of our knowledge, this methodology has never been publicly discussed.

[14] We saw earlier PIE doesn't apply to non network daemons on most distributions yet, for performance reasons

9 Performance considerations

Pmcma starts its analysis by dumping to disk all the mapped section of the analysed binary for easier study. This preliminary phase is particularly costly.

The other phase which is really costly in terms of performance is parsing all sections mapped as writable, list their potential pointers to other sections, and verify if they point to valid assembly instructions by disassembling the destination bytes. The cost of this phase is O(n), where n is the size of writable memory inside the process.

Finally, for each potential function pointer discovered, pmcma will create a new process, overwrite the test pointer with a known value and run the process. This phase is in O(p), where p is the number of potential function pointers discovered.

Experimentally when looking for function pointers, an analysis performed by pmcma ranges from a few seconds when analysing /bin/ping to about one hour when analysing the Opera web browser when crashing after performing a certain amount of heap sparying (resulting in a total of 1.3Gb or memory mapped, among which more than 1.2Gb is writable memory). The average analysis is of several minutes for most network daemons.

It is worth noting that currently, the tests on offsprings are run sequentially one after the other. But in fact, this is not necessary, wether syscall faking is in use or not. In the near future, we hope to modify pmcma to run those tests in parallel instead of running them sequentially.

10 Conclusion

We have briefly presented in this article new exploitation techniques, or sometimes extension of existing ones, and detailed how they could be tested automatically against a target binary vulnerable to invalid memory accesses.

We have exposed how to create exploitation models thanks to a new debugging technique post memory corruption, in order to automatically study the exploity of sub classes of the invalid memory write bug class.

Our proof of concept tool, pmcma, doesn't write exploits itself. Instead, its goal is to analyze all the environment constraints of a given system and provide its user with the best possible attack methodology for a given vulnerability, generalizing many attack vectors and taking into account all the small details (kernel behavior, compiler versions and flags, static and dynamic liking options, set of shared libraries used...) that need be taken into account to write an effective exploit for a given target nowadays. Given the number of tactics available in the literature that work only on very specific occasions (such as specific distributions), the exploitation strategies offered by this tool shall prove valuable to attackers (exploit writers) and software developers or system administrators alike ("is this vulnerability affecting my system or software exploitable by the state of the art of exploitation theory on my particular setup ?").

11 acknowledgements

The author would like to thank in no particular order #busticati, #social, #grsecurity, #ruxcon, #blacksec, THC/TESO, pipacs, spender, twiz, bliss, silvio, andrewg, mercy, gamma, bsdeamon, caddis, izik, xort, redsand, sbz, deadbyte, the grugq, phil, emmanuel, msuiche, the Hackito Ergo Sum (HES) team, the HES Programming comitee, the HES speakers and friends, the /tmp/lab hackerspace, Mark Dowd, Meder Kydyraliev, the CBACert for their technical contributions, ideas and peer reviews. The Toucan System team, his family and his girlfriend, for their encouragements, and their patience.

References

1. PaXTeam: (http://pax.grsecurity.net/docs/aslr.txt)
2. PaXTeam: (http://pax.grsecurity.net/docs/noexec.txt)
3. PaXTeam: (http://pax.grsecurity.net/)
4. AMD: (http://support.amd.com/us/processor_techdocs/24593.pdf)
5. Drepper, U.: (Security enhancements in red hat enterprise linux)
6. Jelinek, J.: (http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html)
7. BBSDaemon: Dynamic program analysis and software exploitation, from the crash to the exploit code. (Phrack magazine)
8. Bania, P.: Spiderpig. Technical report (2008)
9. Cristian Cadar, Daniel Dunbar, D.E.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. Technical report (2008)
10. Thanassis Avgerinos, Sang Kil Cha, B.L.T.H., Brumley, D.: (Aeg: Automatic exploit generation)
11. Henderson, R.: (http://gcc.gnu.org/ml/gcc-patches/2005-05/msg01193.html)
12. localcore, D..: (/bin/su exploit : http://www.exploit-db.com/exploits/209/)
13. Hertz: (at_exit() local exploit)
14. Anonymous: Runtime process infection. (Phrack magazine)
15. Stealth: (http://stealth.openwall.net/local/injectso-0.52.tgz)
16. Brossard, J.: Opera : Select size arbitrary null write, tssa-2011-02, cve-2011-1824 (2011)
17. Intel: Intel 64 and ia-32 architectures software developer's manual. In: Volume 3A: System Programming Guide. (2008)
18. van de Ven, A.: (Limiting buffer overflows with execshield)
19. Mark Daniel, Jake Honoroff, C.M.: (Engineering heap overflow exploits with javascript)
20. Chen, Y.: (Using information leakage to avoid aslr+dep)
21. Vreugdenhil, P.: (Internet explorer 8 on windows 7 exploit form the pown2own contest 2010)