BEYOND FUZZING

Exploit Automation with PMCMA

Jonathan Brossard
Security Researcher & CEO at Toucan System

Say you've been fuzzing a given application, possibly yours, for a few days. You are now left with a bunch of fuzz files that can trigger bugs inside the application. Now what? Send all this data to the vendor (or fix them yourself)? They probably won't even care. What you need to do now is determine which of those bugs are exploitable, with which probability, and then write proper PoCs to demonstrate your claims.

Of course, it is not 1998 anymore and this is by far the hardest part : it requires extensive knowledge of assembly and reverse engineering, encyclopedic knowledge of exploitation techniques & security features bypass.

End of all hopes? Not quite... In fact, we have automated most of the task for you...

Exploitation is hard: overview of software security counter measures

Welcome in 2011: most operating systems now feature non executable memory pages either via software emulation (PaX and its derivatives) or hardware based (Intel NX bit). Most OSes actually enforce X^W meaning that you can't execute writable data: the good old days of putting shellcode in the stack or heaps are over.

Most, if not all sections are randomized, meaning they are mapped at different addresses at runtime.

Heap chunks are also now protected by safe unkinking on both GNU/Linux (ptmalloc) and Windows. This killed entire classes of vulnerabilities such as simple double free().

The stack is most of the time protected by compilers enhancements (/GS compilation under Visual Studio, stack canaries under gcc since version 4.2). In fact, the whole toolchains have been enhanced to reorganize binary sections so that writable data sections, potentially subject to overflows, are not followed by critical sections (such as the Global Offset Table under GNU/Linux). Even the dynamic linking process has been enhanced to minimize attack surface by allowing relocations to be performed at load time, and subsequently remapping the GOT as read only. Hence preventing its malicious hijacking entirely.

Finally, known function pointers such as destructors (stored in the .dtors section when the binary has been compiled with gcc) can be removed entirely via custom linker scripts (removing the entire .dtors section !).

Under those conditions, triggering a bug is by far the easiest part of exploitation. Understanding how to actually exploit the binary, in other words, defining an exploitation strategy, has become the meat of binary hacking.

In the rest of this article, we will focus on the x86 GNU/Linux architecture. PMCMA is also constantly being ported to new architectures, please visit *http://www.pmcma.org* for more details. The actual distribution used to perform the tests in this article is a x86 Ubuntu version 10.10, but Pmcma runs on x86_64 cpus too, and Arch Linux, Debian, Gentoo and Fedora distrubutions have been used successfully with it.

Introducing PMCMA

PMCMA stands for Post Memory Corruption Memory Analysis. In a nutshell, it is a new type of ptrace() based debugger we presented at the latest Blackhat US Conference. PMCMA is free software. It is available at *http://www.pmcma.org/* under the Apache 2.0 license.

Unlike standards debuggers, build by software maintainers to help manually fix software, PMCMA is an offensive one, designed with automation and exploitation in mind.

The core novelty of PMCMA is to allow a debugged process to be replicated at will in memory by forcing it to fork. By creating many replicas of the same process, it allows for easy empirical automation and manipulation. For instance, it can be used to overwrite sequentially all writable sections of memory with a remarkable value after a memory corruption bug has occurred inside the address space, and artificially continue execution. This is the best known way to determine all the function pointers actually called within a binary path. Without the need of lengthy single stepping. And fully automatically.

Determining exploitability with PMCMA

When they are not caught by security checks withing the heap allocator or stack cookie integrity checks, most bugs eventually trigger an invalid memory access resulting in a Segmentation Fault (Signal 11).

There are three types of invalid memory accesses depending no the faulting assembly instruction triggering this access (read mode, write mode or execution mode).

Determining why an application generated an invalid memory access at assembly level is the first step towards exploitation.

Let's use CVE-2011-1824 as an example. It is a vulnerability in the Opera web browser we responsibly disclosed earlier this year[1].

In order to determine what happens at binary level when triggering the vulnerability, let's execute Opera inside a pmcma session. This can be done with a command line such as:

```
./pmcma --fptr --segfault -C `which opera` /
tmp/repro.html
```

Here is an output of the analysis automatically generated by Pmcma:

```
--°=[ Exploitation analysis performed by
PMCMA ]°=--
      1.0  // http://www.pmcma.org
(...)


--[ Command line:
/usr/lib/opera/opera /tmp/repro.html

--[ Pid:
11112

--[ Stopped at:
mov dword ptr [ebx+edx], eax

--[ Registers:
eax=0x00000000
ebx=0x77838ff8
ecx=0x0000001d
edx=0x00000008
esi=0x5d1d4ff8
edi=0x00368084
esp=0xbfeac3ac
ebp=0xbfeac3b8
eip=0x080baceb

--[ Walking stack:
 --> Stack was likely not corrupted (43
valid frames found)

--[ Instruction analysis:
 --> write operation
 --> (2 operands) reg1:edx=0x00000008,reg2:e
ax=0x00000000
 --> the first operand is dereferenced

--[ Crash analysis:
```

** The application received a (SIGSEGV) signal (number 11), while performing an instruction (mov dword ptr [ebx+edx], eax) with 2 operands, of which the first one is being dereferenced.

** The pointer dereference is failing because the register edx, worthing 0x00000008 at this time, is pointing to unmapped memory.

** The impact of this bug is potentially to modify the control flow.

** It is also worth mention that if register eax can only worth 0x00000000 exploitation will be harder (but not necessarily impossible, due to possible unaligned pointer truncations, or by overwriting other data and triggering an other memory corruption indirectly).

The human readable analysis is pretty self explanatory: the faulting instruction didn't corrupt the stack, but Opera generated a Segmentation Fault when executing a « mov » instruction in write mode, potentially allowing an attacker to modify the flow of execution.

This analysis took only a few seconds and contains as much information as you would normally read from an advisory !

In order to turn such a PoC into a working exploit, a shortcoming exists : since we can overwrite some data inside the address space (a few trials and errors quickly ensures that we can in fact write anywhere in the address space), the idea would be to find a function pointer called after this point by the process, and overwrite (or truncate) it to execute arbitrary code.

To balance this example of a potentially exploitable bug, let's have a look at an other analysis, performed on a non exploitable bug :

```
        --°=[ Exploitation analysis
performed by PMCMA ]°=--
                    1.0  // http://www.pmcma.
org

--[ Command line:
/usr/lib/opera/opera /tmp/repro2.html

--[ Pid:
8172

--[ Stopped at:
mov    ebx,DWORD PTR [esi+0x4]

--[ Registers:
eax=0xffffffff
ebx=0x00000031
ecx=0xbf9f3e78
edx=0x00000000
esi=0x00000031
edi=0x0a5badd0
esp=0xbf9fa2b0
ebp=0xbf9f42c8
eip=0x0805a7db

--[ Walking stack:
 --> Stack was likely not corrupted (19
valid frames found)

--[ Instruction analysis:
 --> not a write operation
 --> (2 operands) reg1:ebx=0x00000031
,reg2:esi=0x00000031
 --> the second operand is dereferenced

--[ Crash analysis:
```

** The application received a (SIGSEGV) signal (number 11), while performing an instruction (mov ebx,DWORD PTR [esi+0x4]) with 2 operands, of which the second one is being dereferenced.

** The pointer dereference is failing because the register esi, worthing 0x00000031 at this time, is pointing to unmapped memory.

** The impact of this bug is potentially to perform a controled read operation, leading either to direct information leakage (of an interresting value, or more generally of the mapping of the binary), or indirectly to an other memory corruption bug.

Here, the impact of the bug is much lower since it is essentially a null pointer dereference in read mode : even if he controlled esi entirely, all an attacker could do is assign a value to register eax. In most cases, this is not interesting, unless eax plays a special role in the assembly instructions executed right after this one.

A first possible usage of Pmcma is therefore to determine quickly if a given Segmentation Faulr is of any interest security wise. This is indeed useful for software maintainers as well as computer hackers in general.

Function pointers overwrite

Finding function pointers inside the address space of a process is a complex operation. We could try to disassemble the application including all its libraries and look for explicit instructions such as :

```
call eax
```

This would certainly give us a list of some function pointers inside the address space. But, we don't want to overwrite just about any function pointer: it has to be one actually called during the execution of Opera given the PoC we give it as an input.

A second idea would be to single step execution until we find a suitable function pointer. In this case, given the size of the application, it is clearly unpractical!

This is where Pmcma really becomes handy : it is capable of listing all the function pointers executed after a given point in time, in all of the binary (including its shared library). In this case, the full analysis of Opera with Pmcma takes a few hours.

Listing function pointers

CVE-2010-4344 is a heap overflow in Exim[2]. This bug is interesting for many reasons, in particular because it has been found exploited in the wild in 2010 while it had in fact been reported in 2008.

In a nutshell, Exim before version 4.70 was keeping a buffer in the heap to store data to be sent to its main log file. But it failed at ensuring the buffer wasn't full when adding more data to this buffer, resulting in a heap overflow.

HD More and Jduck wrote a very reliable exploit for this vulnerability by overwriting the configuration file stored in the heap of Exim itself when overwriting this buffer. This is a very elegant solution as it allows them to inject arbitrary shell commands to be executed instead of using shellcodes.

If nonetheless we wanted to use shellcodes instead, we would first need to determine the address of a function pointer stored in the heap (after the address of overflowed buffer) and overwrite it with any chosen address. If the heap itself is executable, a possible option is to return to the buffer itself (which contains user controlled data, hance possibly a shellcode), provided the address of this buffer can be guessed. Since we can send large amount of data (Jduck used 50Mb of padding in the Metasploit exploit for instance), we could still use it as nop sled padding, and bruteforce a bit the address of the heap.

Remember that by definition, a function pointer is stored in a writable section and points to an executable section. It should even point to the beginning of a valid assembly instruction, and very likely to a function prologue. This heuristic is very time saving when listing potential function pointers by parsing a writable section, hence Pmcma normally uses it for its analysis, relaxing it only if it fails to find any suitable function pointer (see next section for an exemple).

Let's look at a snipped of the analysis provided by Pmcma when the debugger is used to attach to the pid of the running Exim :

```
        --°=[ Exploitation analysis
performed by PMCMA ]°=--
                  1.0   // http://www.pmcma.
org

--[ Command line:
/usr/sbin/exim4 -bd -q30m

--[ Pid:
5958
 ...


--[ Loop detection:
<*> crash in a loop : no

--[ Validating function pointers (strict
mode):
  <*> Dereferenced function ptr at
0x080e5000 (full control flow hijack)
     0x080e5000 --> 0xb7463260  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e5048
(full control flow hijack)
     0x080e5048 --> 0xb74e7300  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e504c
(full control flow hijack)
     0x080e504c --> 0xb742d820  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e5064
(full control flow hijack)
     0x080e5064 --> 0xb748d130  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e5108
(full control flow hijack)
     0x080e5108 --> 0xb745fba0  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e5138
(full control flow hijack)
     0x080e5138 --> 0xb745f6d0  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e51a8
(full control flow hijack)
     0x080e51a8 --> 0xb74e6ba0  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e51ec
(full control flow hijack)
     0x080e51ec --> 0xb74632b0  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e5220
(full control flow hijack)
     0x080e5220 --> 0xb74c19e0  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e5228
(full control flow hijack)
     0x080e5228 --> 0xb74c3480  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e5240
(full control flow hijack)
     0x080e5240 --> 0xb74e6f70  //
repeatability:100/100

 <*> Dereferenced function ptr at 0x080e5b88
(full control flow hijack)
     0x080e5b88 --> 0x08097dd4  //
repeatability:100/100

 <*> Dereferenced function ptr at 0xb755c00c
(full control flow hijack)
     0xb755c00c --> 0xb7473ed0  //
repeatability:3/100

 <*> Dereferenced function ptr at 0xb755c018
(full control flow hijack)
     0xb755c018 --> 0xb7473df0  //
repeatability:3/100

 --> total : 14 validated function pointers
           (and found 0 additional control
flow errors)
```

In this case, Pmcma has found 14 potential function pointers with this analysis. Overwriting one of them (actually, any present in the heap) would allow us to modify the flow of execution.

The astute reader will have noticed the repeatability metric provided along with every result: it quantifies the probability to find the associated pointer at this address in memory between different runs (because of ASLR). Those in the data sections of the binary itself (which wasn't compiled as a Position Independent Executable in this case) are always mapped at the same address (100% repeatability). Those in the heap of Exim or in the data sections of shared libraries have a much lower probability of being mapped at the same address between runs (below 3% repeatability).

Targeting function pointers with higher probabilities of being mapped at a given address will lead to much better exploits, requiring less, if any, bruteforcing in general. In our case, because we are studying an overflow instead of an atomic write, we don't care about their address in memory, just their offset from the beginning of the buffer : any function pointer in the heap from the list above would do... unfortunatly, if we look further at the output of Pmcma, we can verify that those two pointers at address 0xb755cXX are in fact part of the data section of the libc, not in the heap :

```
--[ Listing writable sections:
 <*> Section at 0x080e5000-0x080e9000 (RW) /
usr/sbin/exim4
 <*> Section at 0x080e9000-0x080eb000 (RW)
 <*> Section at 0x09051000-0x09074000 (RW)
[heap]
 <*> Section at 0xb73e7000-0xb73e9000 (RW)
 <*> Section at 0xb7400000-0xb7401000 (RW) /
lib/libpthread-2.12.1.so
 <*> Section at 0xb7401000-0xb7403000 (RW)
 <*> Section at 0xb755c000-0xb755d000 (RW) /
lib/libc-2.12.1.so
 <*> Section at 0xb755d000-0xb7560000 (RW)
 <*> Section at 0xb76c1000-0xb76c2000 (RW) /
usr/lib/libdb-4.8.so
 <*> Section at 0xb76e7000-0xb76e8000 (RW) /
lib/libm-2.12.1.so
 <*> Section at 0xb76f2000-0xb76f3000 (RW) /
lib/libcrypt-2.12.1.so
 <*> Section at 0xb76f3000-0xb771b000 (RW)
 <*> Section at 0xb772f000-0xb7730000 (RW) /
lib/libnsl-2.12.1.so
 <*> Section at 0xb7730000-0xb7732000 (RW)
 <*> Section at 0xb7743000-0xb7744000 (RW) /
lib/libresolv-2.12.1.so
 <*> Section at 0xb7744000-0xb7746000 (RW)
 <*> Section at 0xb774b000-0xb774d000 (RW)
 <*> Section at 0xb7758000-0xb7759000 (RW) /
lib/libnss_files-2.12.1.so
 <*> Section at 0xb7763000-0xb7764000 (RW) /
lib/libnss_nis-2.12.1.so
 <*> Section at 0xb776b000-0xb776c000 (RW) /
lib/libnss_compat-2.12.1.so
 <*> Section at 0xb776c000-0xb776f000 (RW)
 <*> Section at 0xb778d000-0xb778e000 (RW) /
lib/ld-2.12.1.so
 <*> Section at 0xbfc27000-0xbfca9000 (RW)
[stack]
```

Advanced usage of Pmcma

Now that the reader is hopefully familiar with the basic strategy followed by Pmcma, let's look at more advanced exploitation strategies.

Since we didn't find a proper function pointer in the heap, it may be a good idea to look for a pointer in the heap pointing not directly to a function pointer, but to a structure elsewhere in memory (for instance in the data section of Exim itself). If we could overwrite this pointer to structure to point to a fake structure in a location we control, we could have a function pointer under our control dereferenced.

Pmcma also automates this search as part of its analysis :

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

     0xbfc679f8 --> 0xbfc67a38  //
repeatability:100/100
     0xbfc67a38 --> 0xbfc67c38  //
repeatability:100/100

 --> total : 2 function pointers identified
inside structures
```

Pmcma identified two such interesting pointers during its analysis. Unfortunately, given the mapping presented earlier, they are located in the stack, and we won't be able to overwrite them using our heap overflow...

Now, plan B is the violent strategy of attempting to overwrite any writable 4byte address located in data sections, hence relaxing the heuristics explained earlier, and see if we can somehow achieve control flow hijacking:

```
--[ Overwriting any writable address in any
section (hardcore/costly mode):

 <*> Dereferenced function ptr at 0xbfc67964
(full control flow hijack)
     0xbfc67964 --> 0xb746ad5f  //
repeatability:100/100

 <*> Dereferenced function ptr at 0xbfc67990
(full control flow hijack)
     0xbfc67990 --> 0xb746b076  //
```

```
repeatability:100/100

( ...)

 <*> Dereferenced function ptr at 0xb73e76d0
(full control flow hijack)
     0x090616d0 --> 0xb776f414  //
repeatability:3/100

 <*> Dereferenced function ptr at 0xb755c00c
(full control flow hijack)
     0xb755c00c --> 0xb7473ed0  //
repeatability:3/100

(...)

 <*> Dereferenced function ptr at 0xbfc67c3c
(full control flow hijack)
     0xbfc67c3c --> 0x080519ad  //
repeatability:100/100

 --> total : 45 validated function pointers
           (and found 0 additional control
flow errors)
```

If we look carefully, the address at 0x090616d0 is in fact inside the heap: by overwriting it, we can achieve full control flow hijacking! Bingo!!

It is worth noticing that this whole automated analysis took place without any user interaction, in less than 5 minutes. Finding the same information manually using disassemblers and debuggers would have taken days to skilled reverse engineers. At best.

The special case of unaligned read/writes

In some cases, like with the Opera vulnerability introduced earlier, overwritting a function pointer to hijack the flow of execution is not practical. In the Opera bug, the value of eax is not user controlled, and is always null. It means an attacker can in fact write 0x00000000 anywhere in memory. If an attacher used this value to overwrite a function pointer, Opera would later on attempt to execute the address 0x00000000, which is never mapped in userland since kernels 2.6.2[3]. In addition, the value of ebx+edx, corresponding to the destination address of the memory write, is always 4 byte aligned, reducing even more the influence of an attacker over the target application.

When such a difficult situation arises, a last resort strategy is to attempt to truncate unaligned variables in writable sections. Listing those sections is typically hard: the current state of the art is to change the permissions of data sections on the fly to not readable, not writable, not executable, wait for a segmentation fault, understand why the segfault occurred by disassembling the latest instruction and looking at its registers... then remap the section readable/writable, execute one instruction (by setting the trap flag in the EFLAG register). Rince and repeat. Obviously, this process is both slow and painful when performed manually.

Pmcma has a better way to list all the unaligned memory accesses inside a binary, by setting the UNALIGNED flag in the EFLAG register. By doing so, Pmcma will automatically receive a signal 7 (Bus error) when a unaligned access is performed. Hence breaking only on unaligned memory access instead of every data access like with the previous method.

To illustrate this feature, let's monitor all the unaligned memory accesses in the OpenSSH deamon of a Fedora 15 distribution.

We start by verifying that OpenSSH is currently running :

```
[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, we initiate an SSH
connection :

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

Then, back in the first terminal, we attach to the pid of the newly instanciated sshd fork by its pid, giving pmcma the –unaligned additional parameter. We obtain the following log :

```
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
...
```

In which we can verify that at each assembly instruction, one of the operands is unaligned. This technique is both faster and more elegant than using mprotect() repeatedly.

Conclusion

Based on those simple examples, we hope to have convinced the reader of the virtues of exploit automation. Pmcma is capable of achieving in little time tasks that would take the best reverse engineers multiple days to do. Pmcma is a free and open source framework and always a work in progress. Feel free to hack it to perform analysis we couldn't have even thought of, and if you like the result, please send us patches! •

>>REFERENCES

1. http://www.toucan-system.com/advisories/tssa-2011-02.txt Opera, SELECT SIZE Arbitrary null write.
2. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4344 Heap-based buffer overflow in Exim before 4.70
3. https://dev.metasploit.com/redmine/projects/framework/repository/revisions/11274/entry/modules/exploits/unix/smtp/exim4_string_format.rb : Exim <= 4.69 Exploit.