t-defence

Proc Memory Injection: Contribution to Atomic Red Team

#TDefenceBusiness

Malware Lab

Summary

Our Malware Lab

T-Defence Malware Lab daily performs dissection of malware with the aim of timely understanding the technological evolutions of attacks, consolidating the knowledge of necessary to make more effective and faster the process of incidents responding, contributing to spreading information about emerging threats into the expert's community and among its clients.

Malware Lab analysts are continuously engaged in searching and experimenting new analysis tools, for increasing accuracy and scope of action with regard to the proliferation of new evasion and anti-analysis techniques adopted by malware.

The Malware Lab is also committed to the development of proprietary tools for malware analysis and supporting the management and response of incidents.

Besides malware analysis, Malware Lab ideated and implemented an automatic process of extraction of **Indicators of Compromise (IOC)** that is daily run on dozens of new malwares, intercepted in the wide for populating our Knowledge Base.

1. Executive Summary

This document presents the contribution to the open-source repository [1] of the tool Atomic Red team [2], consisting of the implementation of six new atomic tests for the sub-technique of the MITRE ATT&CK [3] T1055.009 [4] (Process Injection: Proc Memory), previously uncovered by the tool.

T1055.009 describes the injection of code into a running process through the /proc/<pid>/mem interface, a pseudo-file that exposes a process's virtual address space as writable. On a misconfigured host, an attacker can open this interface, locate a target memory region, and overwrite it to hijack the victim's execution flow, running code within the context of a legitimate process. At the time this work began, the Atomic Red Team repository provided no tests for this sub-technique, so an organization validating its defenses with the tool had no means of empirically verifying its coverage of the technique.

The six atomic tests, written in C, reproduce this behavior and are organized into three families according to the memory region they target: overwriting the instructions at the process's next-executed address (RIP); overwriting the entries of the Global Offset Table (GOT) so that a library call is redirected to the payload; and overwriting the saved return address on the stack.

The tests were submitted to the Atomic Red Team repository as pull request #3337 [5] and merged into the main branch. The contribution is now part of the upstream library and available to all users of the tool. Being merged upstream, the tests also fall under the project's ongoing maintenance and review, which constitutes an external validation of the work.

Author:

- Federico Angarella

2. Background

Atomic Red Team, as its maintainers state, is "*an open source library of tests designed to test your organization's security controls*". It provides a continuously updated library of atomic tests, each reproducing the behavior of a specific attack technique. Executing a test against a target environment gives empirical evidence of whether the corresponding behavior is detected, allowing the defensive posture to be measured.

T1055.009 Process Injection: Proc Memory falls under the parent technique T1055 (Process Injection) within the Stealth (TA0005) and Privilege Escalation (TA0004) tactics of the MITRE ATT&CK framework. It describes the injection of code into a running process through the /proc/<pid>/mem interface [6], inside the /proc filesystem [7]. It exposes a process's virtual address space as a writable pseudo-file. In an environment that is not correctly configured, an attacker can open this interface, look for a target memory region, and overwrite it to hijack the victim process's execution flow, executing code within the context of a legitimate process.

3. Development

The six tests written in C fall into three families, organized by the memory region of the process they act upon. Together they cover the main ways in which an adversary can exploit the /proc/<pid>/mem interface to inject malicious code. This chapter describes how each test works.

Before discussing the individual tests, there is an important detail common to all of them that must be addressed: why it is possible to write to /proc/<pid>/mem even in executable memory regions. Many of the techniques described below rely on this, yet the W^X (Write XOR Execute) principle would normally prevent it: a page marked executable is not writable, and a direct write attempt through an ordinary MOV instruction would trigger a page fault, causing the kernel to respond with a SIGSEGV.

Access through /proc/<pid>/mem bypasses this constraint. The write is not performed by the CPU in the target process's context, but by the kernel on behalf of the caller, which handles the request as file I/O rather than as an ordinary memory operation. Internally the kernel uses the FOLL_FORCE flag, which allows the write to proceed even on pages lacking write permission. This mechanism exists to support legitimate use cases such as debuggers setting breakpoints. A more in-depth explanation of this mechanism is available in [8].

Test 1 and 2: instruction overwrite in-place and jump

Alongside /proc/<pid>/mem, the /proc directory of a process exposes another pseudo-file relevant here: /proc/<pid>/syscall [9], which reports the registers of the process at the point of the ongoing system call. This family of techniques reads from it the value of the RIP register: the address of the next instruction the process will execute, which in this context works as the re-entry point after the ongoing system call. The instructions located at that address are then overwritten. In these implementations, reading the register is performed by the call get_reg(pid, "rip"), whose pseudocode is shown below.

In the **in-place overwrite variant**, the payload, in the form of shellcode, is written directly starting from the address held in RIP, overwriting the instructions the process would otherwise have executed.

In the **jump overwrite variant**, by contrast, the function find_code_cave() is first invoked to search the victim process's memory for a zeroed executable region. Targeting empty memory ensures that the injector does not corrupt any pre-existing code, which is particularly relevant for stealth variants designed to resume the victim's normal execution once the payload has run. Once a code cave of sufficient size has been identified, the shellcode is written into it, and a stub that jumps to the payload is then placed at the address held in RIP, completing the injection.

```
FUNCTION get_reg(pid, reg):
      // Open the syscall pseudo-file
      fd = open("/proc/{pid}/syscall", O_RDONLY);

      // Parse the buffer into its register fields
      registers = parse(buffer)

      // Return the requested register
      RETURN registers[reg]
END FUNCTION

INSTRUCTION OVERWRITE IN-PLACE:
      // Create a child process to inject:
      // we create one and not inject an existing
      // because it works also in case ptrace_scope >= 1 [10]
      pid = create_victim_child_process()

      // Gather the address of the next instruction executed
      instruction_pointer = get_reg(pid, "rip")

      // Overwrite the next instruction location with our payload instead
      // NOP padding absorbs the kernel's post-syscall RIP restore
      // this mechanism is explained in detail in [11]
      // this resource was also the main source of information for this work
      write_in_mem(instruction_pointer-2, pid, NOP padding + payload, payload_len +2)
END


INSTRUCTION OVERWRITE JUMP:
      // Create a child process to inject:
      // we create one and not inject an existing
      // because it works also in case ptrace_scope >= 1
      pid = create_victim_child_process()

      // Gather the address of the next instruction executed
      instruction_pointer = get_reg(pid, "rip")

      // Locate a zeroed executable region to write our payload in it:
      // scan the memory regions listed in the pseudo-file /proc/<pid>/maps,
      // then read each executable one from /proc/<pid>/mem
      // looking for a run of consecutive 0x00 bytes at least payload_len long
      payload_address = find_code_cave(pid, payload_len)

      // Build a jumper to that region with the NOP padding
      jumper = create_jumper(payload_address)

      // Write our payload in the region found before
      write_in_mem(payload_address, pid, payload, payload_len)

      // Overwrite the next instruction location with our jumper instead
      write_in_mem(instruction_pointer - 2, pid, jumper, jumper_len)
END
```

Test 3: GOT injection

This technique exploits the Global Offset Table (GOT), which, among other things, stores the resolved addresses of the linked shared objects' functions. The idea is as follows: by injecting the shellcode into a code cave, as in the previous technique, and overwriting precise entries of the GOT with the shellcode's address, at the next call of an external function the execution will be redirected to the malicious code instead of the original library function.

Identifying and overwriting the GOT, performed by the function get_got(pid), is not trivial. The goal is to locate the .got.plt [12] section, but in the running process it cannot be found by name. The reason is that section names are not part of the loaded process image: the mapping between a section's name and its location lives in the section header table and in the associated section-name table (.shstrtab), and neither of these is loaded into memory at runtime. The information must therefore be recovered from the original executable on the filesystem, whose path is available through the /proc/<pid>/exe symbolic link. The procedure reads the file's ELF header (Elf64_Ehdr), which gives the offset of the section header table. It then walks the section headers one by one, resolving each section's name through .shstrtab until the .got.plt section is found and its offset obtained.

A final aspect concerns the first three entries of the GOT, which do not contain addresses of library functions but have reserved meaning: the address of the .dynamic section, the pointer to the link_map structure, and the pointer to the _dl_runtime_resolve function of the dynamic linker. These three entries must be excluded from the overwrite, since modifying them would compromise the dynamic resolution of symbols and would cause the process to crash even before the shellcode's execution. All the subsequent entries are instead overwritten with the shellcode's address, so that the payload is executed in place of whatever external function the process attempts to invoke.

```
FUNCTION get_got(pid, out got_size):
     // Open the original executable through the symbolic link
     fd = open("/proc/{pid}/exe", O_RDONLY);

     // Read the ELF header
     ehdr = read ELF header from fd

     // Read the section-header string table (shstrtab),
     // which holds the names of all sections
     shstrtab_hdr = read header from ehdr.e_shstrndx
     shstrtab = read shstrtab_hdr.sh_size bytes from fd at shstrtab_hdr.sh_offset

     // Parse the section headers looking for the one named ".got.plt"
     FOR each section header shdr      // Gathered from fd at ehdr.e_shoff
          name = shstrtab + shdr.sh_name
          IF name = ".got.plt":
               got_addr = shdr.sh_addr
               got_size = shdr.sh_size
               RETURN got_addr
     RETURN -1
END FUNCTION



GOT INJECTION:
     // Create a child process to inject:
     // we create one and not inject an existing
     // because it works also in case ptrace_scope >= 1
     pid = create_victim_child_process()

     // Locate a zeroed executable region to write our payload in it:
     // scan the memory regions listed in the pseudo-file /proc/<pid>/maps,
     // then read each executable one from /proc/<pid>/mem
     // looking for a run of consecutive 0x00 bytes at least payload_len long
     payload_address = find_code_cave(pid, payload_len)

     // Write our payload in the region found before
     write_in_mem(payload_address, pid, payload, payload_len)

     // Gather GOT address and size
     got_address = get_got(pid, out got_size)

     // Build a block that repeats the payload address once per entry:
     // skip the first 3 reserved entries (+24 bytes, 3 * 8), and fill
     // each of the remaining 8-byte slots with the payload address.
     entry_count = (got_size - 24) / 8
     got_payload = [ payload_address ] * entry_count

     // Overwrite every imported-function entry with the payload address
     write_in_mem(got_address + 24, pid, got_payload, got_size - 24)
END
```

Test 4, 5 and 6: Return Address Overwrite

The remaining three tests are return address overwrite variants, all of which exploit the saved return address on the stack. The first one writes the payload into an executable region and sets its address as the return address. The second writes the payload into a writable, non-executable (rw-) region and injects a ROP chain into the return address that first calls mprotect to mark the region executable (r-x), then transfers execution to the payload as in the first variant. The third performs the malicious activity entirely through a ROP chain, without injecting a separate payload.

The first basic version works much like the earlier jump instruction overwrite: the payload is written as shellcode into a code cave, and what changes is how execution is redirected to it. The saved return address on the stack is replaced with the address of the code cave now holding the malicious code, so that when the current function returns, the execution flow is diverted to the payload.

```
RETURN ADDRESS OVERWRITE:
     // Create a child process to inject:
     // we create one and not inject an existing
     // because it works also in case ptrace_scope >= 1
     pid = create_victim_child_process()

     // Locate a zeroed executable region to write our payload in it:
     // scan the memory regions listed in the pseudo-file /proc/<pid>/maps,
     // then read each executable one from /proc/<pid>/mem
     // looking for a run of consecutive 0x00 bytes at least payload_len long
     payload_address = find_code_cave(pid, payload_len)

     // Write our payload in the region found before
     write_in_mem(payload_address, pid, payload, payload_len)

     // Obtain the stack pointer, needed to locate the return address
     stack_pointer = get_reg(pid, "rsp")

     // The return address cannot be located deterministically on the stack,
     // so a set of candidate slots is considered and each is overwritten
     // with the payload address.
     overwrite_possible_ret_addrs(pid, stack_pointer, payload_address)
END
```

One of the advantages of /proc/<pid>/mem is the ability to write to executable memory without invoking mprotect, a syscall which can act as an indicator of compromise (IoC). Writing to executable memory can, however, still be detected. The mprotect variant was implemented to offer a complementary approach. Here the payload is injected into a region that is initially writable and non-executable, avoiding any direct write to executable memory at the cost of reintroducing mprotect. When the return address is overwritten, before execution returns to it, a ROP chain invokes mprotect to change the region's permissions from rw- to r-x, after which control is transferred to the payload.

The ROP chain is not written directly at the return-address slots but into a separate writable, zeroed region that works as a fake stack. Placing it at those slots would fail: since all the candidates are overwritten at once, and typically lie close together, the copies of the chain would overlap and corrupt one another. The chain is therefore written only once, into the fake stack, while each candidate slot receives a sixteen-byte trampoline, composed of the pivot gadget pop rsp; ret followed by the address of the fake stack. When the function returns, the gadget loads that address into the stack pointer, so that RSP now points at the fake stack; from there each subsequent ret walks through the chain, executing the gadgets in sequence.

```
RETURN ADDRESS OVERWRITE MPROTECT ROP CHAIN:
     // Create a child process to inject:
     // we create one and not inject an existing
     // because it works also in case ptrace_scope >= 1
     pid = create_victim_child_process()

     // Locate the pivot gadget to build the trampoline later
     pivot_gadget = find_gadgets(pid, "pop rsp; ret;")

     // Locate a zeroed writable region large enough to hold both the payload
     // and the ROP chain (which is placed right after the payload):
     // scan the memory regions listed in the pseudo-file /proc/<pid>/maps,
     // then read each writable one from /proc/<pid>/mem,
     // looking for a run of consecutive 0x00 bytes long enough.
     writable_region = find_zeroed_writable_cave(pid, payload_len +
     rop_chain_size)

     // Write the payload at the start of the region found above.
     write_in_mem(writable_region, pid, payload, payload_len)

     // Build the ROP chain that calls mprotect to make the payload region
     // executable, then transfers control to the payload.
     rop_chain = build_mprotect_rop_chain(writable_region, pid)

     // The ROP chain is not placed at the return address slots directly:
     // since every return-address candidate is overwritten at once, writing
     // the chain there would superimpose overlapping copies and corrupt it.
     // It is therefore written once, right after the payload.
     rop_chain_address = writable_region + payload_len
     write_in_mem(rop_chain_address, pid, rop_chain, rop_chain_size)

     // Obtain the stack pointer, needed to locate the return address.
     stack_pointer = get_reg(pid, "rsp")

     // build the trampoline to be placed at each return-address candidate
     trampoline[0] = pivot_gadget
     trampoline[1] = rop_chain_address

     // The return address cannot be located deterministically on the stack,
     // so a set of candidate slots is considered and each is overwritten
     // with a trampoline which pivots the stack onto the ROP chain
     overwrite_possible_ret_addrs(pid, stack_pointer, trampoline)
END
```

The two previous variants still write executable code into memory, whether by using an existing r-x region or by calling the mprotect syscall on it; the **full ROP chain** removes the need for shellcode entirely. Nothing is written to executable memory and mprotect is never called: only the ROP chain and the argument strings it needs are placed in writable, non-executable memory, and the sole code executed is the legitimate code already mapped in the process. As in the mprotect variant, the chain is written into a fake stack and reached through the trampoline mechanism described above.

```
RETURN ADDRESS OVERWRITE FULL ROP CHAIN:
     // Create a child process to inject:
     // we create one and not inject an existing
     // because it works also in case ptrace_scope >= 1
     pid = create_victim_child_process()

     // Locate the pivot gadget to build the trampoline later
     pivot_gadget = find_gadgets(pid, "pop rsp; ret;")

     // Build the ROP chain that carries out the malicious behavior entirely.
     rop_chain = build_rop_chain_exec_cmd(pid, rop_chain_size)

     // Locate a zeroed writable region large enough to hold the ROP chain
     // scan the memory regions listed in the pseudo-file /proc/<pid>/maps,
     // then read each writable one from /proc/<pid>/mem,
     // looking for a run of consecutive 0x00 bytes long enough.
     writable_region = find_zeroed_writable_cave(pid, rop_chain_size)

     // The ROP chain is not placed at the return address slots directly:
     // since every return-address candidate is overwritten at once, writing
     // the chain there would superimpose overlapping copies and corrupt it.
     // It is therefore written once into the fake stack.
     write_in_mem(writable_region, pid, rop_chain, rop_chain_size)

     // Obtain the stack pointer, needed to locate the return address.
     stack_pointer = get_reg(pid, "rsp")

     // Build the trampoline to be placed at each return-address candidate.
     trampoline[0] = pivot_gadget
     trampoline[1] = writable_region

     // The return address cannot be located deterministically on the stack,
     // so a set of candidate slots is considered and each is overwritten
     // with the trampoline, which pivots the stack onto the ROP chain.
     overwrite_possible_ret_addrs(pid, stack_pointer, trampoline)
END
```

4. References

[1] https://github.com/redcanaryco/atomic-red-team

[2] https://www.atomicredteam.io/

[3] https://attack.mitre.org/

[4] https://attack.mitre.org/versions/v19/techniques/T1055/009/

[5] https://github.com/redcanaryco/atomic-red-team/pull/3337

[6] https://www.man7.org/linux/man-pages/man5/proc_pid_mem.5.html

[7] https://www.man7.org/linux/man-pages/man5/proc.5.html

[8] Linux Internals: How /proc/self/mem writes to unwritable memory - offlinemark

[9] https://www.man7.org/linux/man-pages/man5/proc_pid_syscall.5.html

[10] https://www.kernel.org/doc/html/latest/admin-guide/LSM/Yama.html

[11] https://www.akamai.com/blog/security-research/the-definitive-guide-to-linux-process-injection

[12] https://linuxvox.com/blog/what-is-the-difference-between-got-and-got-plt-section/#deep-dive-the-got-plt-section

t-defence

Next | Donexit | Foramil | Innodesi

Via Giacomo Peroni, 452 – 00131 Roma
tel. 06.45752720 – info@defencetech.it
www.t-defence.it

#TDefenceBusiness