

Going beyond /etc/shadow

Alexandra Sandulescu, Matteo Rizzo

Google

Can we “get root”

- Threat model
 - Published PoC: unprivileged user attacking the host kernel
 - Real world scenario: sandboxed or running in VM
- Leak rate
 - Published PoC: 1kB/s
 - Real world scenario
 - total number of leaked bytes needed to “get root”
 - fast
- Risk
 - Published PoC: leaks the root password hash
 - Real world scenario: arbitrary memory read primitive
 - keys or tokens that if leaked, break the security guarantees of the system
- Limitation
 - Published PoC: dependent on the host kernel image, gadgets, exploit primitives location in memory
 - Real world scenario: flexible :)

01

Better attack scenarios

Sandboxed attack scenario

- Most PoCs attack from an unsandboxed unprivileged process
- Reality: kernel exploits are much more practical in this scenario
 - <https://github.com/google/security-research/tree/master/pocs/linux/kernelctf>
- Better: focus on sandboxed attackers
 - E.g. Chromium sandbox, sandbox2, sandboxed API

Sandboxed attack scenario

- Sandboxes restrict which system calls an attacker has access to
- Bad for kernel exploits (can eliminate most attack surface)
- A realistic retbleed/inception exploit only needs 2-3 common system calls
- Sandboxed services make CPU exploits more appealing

On leaking /etc/shadow

- A lot of POCs focus on leaking the contents of /etc/shadow

- The `prctl` function can be used to set the `no_new_privs` attribute. For example:

```
int prctl(PR_SET_NO_NEW_PRIVS, 1L, 0L, 0L, 0L);
```

DESCRIPTION [top](#)

- Set the calling thread's `no_new_privs` attribute. With `no_new_privs` set to 1, `execve(2)` promises not to grant privileges to do anything that could not have been done without the `execve(2)` call (for example, rendering the set-user-ID and set-group-ID mode bits, and file capabilities non-functional).
- Once set, the `no_new_privs` attribute cannot be unset. The setting of this attribute is inherited by children created by `fork(2)` and `clone(2)`, and preserved across `execve(2)`.
- You still have to crack the password

On leaking /etc/shadow

- Scanning memory for /etc/shadow is not realistic
- Most exploits leak at best a few 10s of KB/s
- We have servers with many TiB of memory.
 - Scanning all of that would take many weeks.

Alternatives to reading /etc/shadow

- Credentials that an attacker can actually use. Some ideas:
 - Leak cookies from Chrome
 - Leak TLS private keys from a web server
 - Leak access tokens from another service that interacts with an API

Alternatives to scanning memory

- Targeted leaking is much better with a low-bandwidth attack
- Traverse the Linux process tree to find a target
- Parse the target process's VMA tree to find its stack/heap
- Parse the page tables to resolve virt->phys mappings

Targeted leaking on Linux

```
struct task_struct init_task;
```



```
struct task_struct {  
    // ...
```

```
    struct mm_struct    *mm;  
    struct list_head    children;  
    struct list_head    tasks;
```

```
    // ...
```

```
};
```

```
struct mm_struct {  
    // ...
```

```
    pgd_t * pgd;
```

```
    // ...
```

```
    unsigned long start_code, end_code, start_data, end_data;  
    unsigned long start_brk, brk, start_stack;  
    unsigned long arg_start, arg_end, env_start, env_end;
```

```
};
```

Root page table

Start of stack/heap/argv, ...

Process tree

List of all tasks

02

Faster exploits

Training in speculation

Retbleed/inception exploit: `jmp kernel_addr`

- Slow (1 page fault + SIGSEGV / training)
- Even worse with ptrace (1 roundtrip to the tracer / training)
- Noisy
- Context switches evict cache/TLB/branch predictor entries

Training in speculation

Better: train in speculation

- Fast (no architectural page faults)
- Ptracer not notified
- Completely silent
- No context switches

```
clflush [rsp-8]  
mfence  
call after
```

jmp training

```
after:  
add rsp, 8  
ret
```

Speculative ROP

7.2 Using non-trivial disclosure gadgets

The gadgets that we discover are non-trivial to exploit. We discuss some of the problems that we encounter and how we overcome them.

- From the retbleed paper:
- Usually hard/impossible to find good gadgets even in large binaries (e.g. kernel)
- Idea: chain multiple simple gadgets to do something more powerful

Speculative ROP: flush+reload

The diagram illustrates a speculative ROP attack using a 'flush+reload' technique. It shows two code snippets on the left and two on the right, connected by red arrows indicating control flow. The left snippets represent the original code, and the right snippets represent the state after a speculative ROP chain execution. The first snippet on the left is a function prologue for `__ksys_mprotect`, followed by an ellipsis and a `ret` instruction. A red arrow points from this `ret` to the first snippet on the right, which contains `movzx eax, BYTE PTR [rdi]`, `pop rbp`, and `ret`. A second red arrow points from the `ret` of the first right snippet to the second snippet on the left, which contains `shl rax, 12`, `pop rbp`, and `ret`. A third red arrow points from the `ret` of the second left snippet to the second snippet on the right, which contains `movzx eax, WORD PTR [rsi+rax]`.

```
__ksys_mprotect:  
    [...]  
    ret
```

```
movzx eax, BYTE PTR [rdi]  
pop rbp  
ret
```

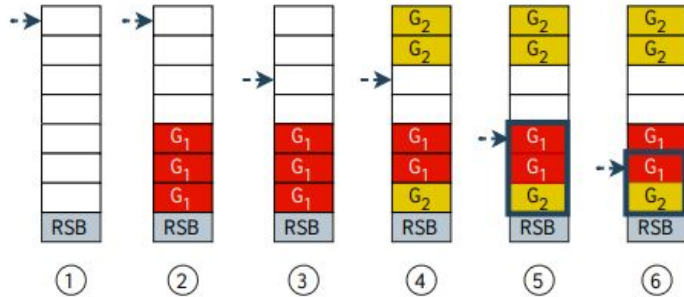
```
shl rax, 12  
pop rbp  
ret
```

```
movzx eax, WORD PTR [rsi+rax]
```

Trained return target



Speculative ROP: Inception



- Dueling recursive phantom calls
- The attacker doesn't control the order
 - G1 must be idempotent irt G2 state

Source: https://comsec.ethz.ch/wp-content/files/inception_sec23.pdf

Speculative ROP: Gadget chaining in the RAS

```
rdtsc
nop
nop
nop
shl    rdx,0x20
or     rdx,rax
mov    rax,QWORD PTR [rip+0x3069772]
cmp    rdx,rax
jb     ffffffff81059ff4 // Phantom JMP
```


RAS
G1
G2
G1
G2
...

```
G1 - 5: // Phantom Call G2
G1: pop
pop
..
ret
```

```
G2 - 5: // Phantom Call G1
G2: movzx eax, BYTE PTR [reg]
movzx eax, WORD PTR [rbx+rax*4]
```

Inception: Speculation control

the recovery mechanisms, as shown in Figure 6. Specifically, on Zen 3 microarchitectures we hijack a single return ~~instruction~~ by first exhausting 17 uncorrupted RSB entries. On Zen 4, we need to exhaust 8 uncorrupted RSB entries, after which we control the next 16 return target predictions. We find that the number of RSB entries polluted heavily relies ~~on the exact location~~ at which we trigger PHANTOM speculation, the state of the cache, the state of the BTB, and the preceding control flow.



Is this even exploitable?



Improve the gadget?

Source: https://comsec.ethz.ch/wp-content/files/inception_sec23.pdf

Inception DSI

```
cpuid // Dispatch serializing instruction
instr
instr
..
ret // At this point the RAS is fully controlled
```

- AMD Security Bulletin:
<https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7031.html>
- No new information, previous guidance applies

03

Impact

What we learned

- Real world exploitation incurs extra challenges
 - complex threat model: sandbox-host, guest-host
 - restrictive environment: can't spawn threads, processes, can't co-locate, can't crash, etc.
 - gadgets limitations
 - widely used side channels might not be available
- Risk is sometimes misunderstood
- More exploits would be nice



HwCTF

- Part of Google VRP
- Reward **exploits** of CPU bugs for a specific **threat model**
 - Guest-Host arbitrary memory read
 - Leak something interesting e.g. data that belongs to other processes or to VMs running on the same host
 - Fast and stable
- Some mitigations are disabled
 - We care a lot about exploit techniques
- First release
 - BHI is in scope
- Not live yet 🙄
 - We can share the kernel image and the host configuration

Questions