Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying

Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes

Georgia Tech, CISPA, Saarland University, MPI-SWS, DFKI
Starting with the famous brother: Use-after-free

```c
char* foo = (char*) malloc (100);
*foo = "abc";
free (foo);

printf(%s, *foo);
```

*foo = ???
Starting with the famous brother: Use-after-free

```c
char* foo = (char*) malloc (100);
*foo = "abc";
free (foo);
char* bar = (char*) malloc (100);
*bar = "evil";
printf(%s, *foo);
```

*foo = "evil"
Uninitialized use vulnerabilities on the stack

Vulnerable Function:

```c
struct A *foo;
...
foo->complete();
```
Uninitialized use vulnerabilities on the stack

Setup function:

```c
int buf[50];
for (i=0; i<50; i++) {
    buf[i] = evil;
}
```

Vulnerable Function:

```c
struct A *foo;
...
foo->complete();
```

```
Stack
...
evil
evil
evil
evil
...
```

```
evil->complete()
```
Uninitialized uses pose critical security risks

1. Setup Function: Control uninitialized variable

2. Vulnerable Function: Uninitialized Use

Overlapping Memory

Arbitrary read, write, and execution
In reality, uninitialized-use problems are overlooked

• Uninitialized uses were regarded as undefined behaviors

2015-2016: 16 Linux kernel patches, 1 CVE

• Full memory safety techniques exclude uninitialized uses as a target

• Widespread belief: uninitialized memory is uncontrollable
Manually exploiting uninitialized uses in the Linux kernel stack is difficult

Overlapping Memory

Setup function

Vulnerable function
Manually exploiting uninitialized uses in the Linux kernel stack is difficult.

Diagram:
- Overlapping Memory
- Setup function
- Vulnerable function
- 300 different syscalls

NDSS 2017
Manually exploiting uninitialized uses in the Linux kernel stack is difficult

Overlapping Memory

Setup function

Vulnerable function

300 different syscalls

Different parameter values for each syscall
Primary kernel stack usage

• 90% of all system calls only use the first 2KB of the kernel stack
• Most interesting region to target
Targeted stack spraying

- Deterministic stack spraying
- Exhaustive memory spraying

2KB
8 or 16 KB stack
Targeted stack spraying

Deterministic stack spraying

Exhaustive memory spraying

2KB

8 or 16 KB stack
Deterministic stack spraying overview

- **Symbolic Execution**: Explore execution paths of syscalls
- **Parameters**:
- **Dynamic Verification**: Run and verify stack spraying
Path exploration using Symbolic Execution

```c
syscall(int p1, int p2, char* buf)
```

Path 1: \( p1 < 5; \ p2 < p1 \)

Path 2: \( p1 < 5; \ p2 \geq p1 \)

Path 3: \( p1 \geq 5 \)
SE: handling path explosion due to unbounded loops

Fuzz loop variable (p2) in dynamic verification phase

Path 3: \( p1 \geq 5 \)

\( \text{LOOP}(p2) \)
Verify stack spraying by executing paths

Path 1: p1<5; p2<p1
syscall(p1=3, p2=2, buf="UUID")
Verify stack spraying by executing paths

Path 2: p1<5; p2>=p1
syscall(p1=3, p2=3, buf="UUID")
Verify stack spraying by executing paths

Path 3: \( p1 \geq 5 \)
\[ \text{syscall}(p1=6, \text{FUZZ}(p2), \text{buf}="\text{UUID}" ) \]
Mapping syscalls and parameters to memory locations

syscall(p1=6, p2=4, buf="UUID")
syscall(p1=6, p2=3, buf="UUID")
syscall(p1=6, p2=2, buf="UUID")
syscall(p1=6, p2=1, buf="UUID")
Automatically exploiting uninitialized uses in the Linux kernel is possible!

\[
\text{syscall}(p1=6, \ p2=4, \ \text{buf}="\text{UUID}\"
\)
Achieved Linux kernel stack coverage

Deterministic Stack Spraying: 39% of 2KB
Deterministic Stack Spraying + Exhaustive Memory Spraying: 91% of full stack
Real world case study: Linux Kernel Privilege escalation CVE-2010-2963

• CVE-2010-2963: Uninitialized pointer used for write
• Found by Kees Cook

• Setup: get_video_tuner32  →  Vuln: get_microcode32

• We automatically found 27 syscalls that can control the uninitialized pointer
Efficient mitigation by zero-initialization

```
struct A *foo;
...
foo->complete();
```

```
struct A *foo;
...
bar(foo);
```

Identify unsafe pointer-type fields
Efficient mitigation by zero-initialization

```
struct A *foo = 0;
...
foo->complete();

struct A *foo = 0;
...
bar(foo);
```

LLVM IR → Identify unsafe pointer-type fields → Zero-initialize all these fields → Secured IR
Mitigation performance overhead

• Syscall performance overhead with LMBench
  • Average: 1.95%

• User program performance overhead with SPEC benchmarks
  • Average: 0.47%
Conclusions

• Uninitialized stack variables can be reliably controlled

• Uninitialized use is a critical attack vector

• Memory-safety techniques should include uninitialized use as a prevention target