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 Institute of Technology
[§]DFKI, [¶]MPI-SWS, [‡]CISPA, Saarland University
Saarland Informatics Campus
{kjlu, wenke}@cc.gatech.edu, {walter, pfaff, backes}@cs.uni-saarland.de, stefan.nuernberger@dfki.de

***Abstract*—A common type of memory error in the Linux kernel is using uninitialized variables (uninitialized use). Uninitialized uses not only cause undefined behaviors but also impose a severe security risk if an attacker takes control of the uninitialized variables. However, reliably exploiting uninitialized uses on the kernel stack has been considered infeasible until now since the code executed prior to triggering the vulnerability must leave an attacker-controlled pattern on the stack. Therefore, uninitialized uses are largely overlooked and regarded as undefined behaviors, rather than security vulnerabilities. In particular, full memory-safety techniques (e.g., SoftBound+CETS) exclude uninitialized use as a prevention target, and widely used systems such as OpenSSL even use uninitialized memory as a randomness source.**

In this paper, we propose a fully automated targeted stack-spraying approach for the Linux kernel that reliably facilitates the exploitation of uninitialized uses. Our targeted stack-spraying includes two techniques: (1) a deterministic stack spraying technique that suitably combines tailored symbolic execution and guided fuzzing to identify kernel inputs that user-mode programs can use to deterministically guide kernel code paths and thereby leave attacker-controlled data on the kernel stack, and (2) an exhaustive memory spraying technique that uses memory occupation and pollution to reliably control a large region of the kernel stack. We show that our targeted stack-spraying approach allows attackers to reliably control more than 91% of the Linux kernel stack, which, in combination with uninitialized-use vulnerabilities, suffices for a privilege escalation attack. As a countermeasure, we propose a compiler-based mechanism that initializes potentially unsafe pointer-type fields with almost no performance overhead. Our results show that uninitialized use is a severe attack vector that can be readily exploited with targeted stack-spraying, so future memory-safety techniques should consider it a prevention target, and systems should not use uninitialized memory as a randomness source.

I. Introduction

In programming languages such as C and C++, programmers decide whether to initialize a variable with a deterministic value when it is allocated. C enthusiasts often argue that if programmers know that the code will later set a proper value anyway, initialization on allocations is an unnecessary use of precious CPU cycles. This argument makes sense from a functional point of view since such an unnecessary use of CPU cycles can cause a significant runtime overhead if it occurs up to millions of times per second, as it does in programs such as in OS kernels. However, manually keeping track of all possible code paths to ensure proper initialization is an error-prone task. Even worse, automatic detection of uninitialized use, such as the warning of compilers, is inaccurate for several reasons. First, inter-procedural tracking often leads to false positives and false negatives because of problems such as aliasing. Second, whether an uninitialized-use warning is justified is highly subjective: While some programmers may prefer a warning in every possible case, others might consider a warning unnecessary if it would not cause an observable error or is likely a false positive.

Uninitialized data represents arbitrary values that were coincidentally stored in the memory. If the uninitialized data is used for control flow, such as the case in which an uninitialized function pointer is dereferenced, the execution of the program or even the kernel can potentially be hijacked. A recent example of that control flow hijacking is caused by uninitialized use is shown in Figure 1. Here, the pointer `backlog`, defined at line `7`, is not initialized in a code path that can be triggered only by special inputs (i.e., when `cpg->eng_st != ENGINE_IDLE`), which is dereferenced at line `15`. An attacker can exploit such an uninitialized-use vulnerability to achieve arbitrary code execution by controlling the value of `backlog`, such as making `backlog` point to a function pointer to malicious code.

Despite their potentially dangerous consequences, uninitialized-use bugs are very seldom classified as security vulnerabilities [16, 49], which arguably originates from the perception that it is hard for an attacker to control the memory layout in order to make dereferencing exploitable. In particular, widely used systems such as OpenSSL explicitly use uninitialized data for the generation of entropy (see function `ssleay_rand_bytes()` in the `SSLeay` implementation) and hence ground their security on the assumption that such data is impossible to control or predict. On the other hand, our study revealed that in 2015 and 2016 alone, although 16 uninitialized-use vulnerabilities have been patched in the Linux Kernel, only *one* was reported for a CVE. In fact, since 2004, only eight uninitialized-use vulnerabilities in the Linux kernel have been reported for a CVE. For example, the severe uninitialized-use vulnerability shown in Figure 1 has not been reported for a CVE. From a security point of view, uninitialized use or more precisely, temporal memory errors should be included as a prevention target in state-of-the-art memory protection mechanisms. However, advanced security mechanisms such as SoftBound+CETS [30, 31] and

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23387

```c
/* file: drivers/crypto/mv_cesa.c
 * uninteresting code lines are omitted
 */
static int queue_manag(void *data)
{
    /* back log is defined without initialization */
    struct crypto_async_request *backlog;

    if (cpg->eng_st == ENGINE_IDLE) {
        backlog = crypto_get_backlog(&cpg->queue);
    }

    if (backlog) {
        /* uninitialized pointer dereferenced! */
        backlog->complete(backlog, -EINPROGRESS);
    }

    return 0;
}
```

Fig. 1: A recent uninitialized pointer dereference vulnerability discovered in the Linux kernel and patched in April 2015. `backlog`, a pointer that is not initialized if `cpg->eng_st != ENGINE_IDLE`, is dereferenced later on. Therefore, arbitrary code execution occurs if an attacker can control the value of `backlog` on the kernel stack.

WatchdogLite [29]), which claim full memory safety, do not currently cover uninitialized uses.

A. Challenges

Most uninitialized uses stem from the stack rather than the heap: Out of the 16 aforementioned uninitialized-use variables in the Linux kernel, 11 variables are stored on the stack. In contrast to uninitialized memory on the heap, that on the stack is hard to control for several reasons:

1) Stack memory is frequently and unpredictably reused by other parts of code; hence, prepared data on the stack is likely to be overwritten by other data.
2) The size of the stack objects is usually small and fixed, so stack control is inherently challenging.
3) Stack depth (especially for the kernel) is strictly checked, so the broad control of the stack is challenging.

As a result, to control the value of an uninitialized variable, any successful attack needs to overcome these three challenges. To overcome these challenges, we need to fulfill three requirements:

R1: The relative address of the uninitialized variable inside the stack must be known.

R2: The memory at the discovered address of R1 must be controllable. That is, we can write arbitrary data to this memory.

R3: Data written in R2 must not be overwritten before it is used by the vulnerable function that suffers from an uninitialized use.

Until now, fulfilling all three requirements has constituted a manual and labor-intense task if it succeeds at all. In the past, successful exploits relied on other memory-corruption vulnerabilities to fulfill requirement R2, or they were simply crafted in an unprincipled manner. For example, as we will show in §VI-F, Cook [12] found that the syscall with the uninitialized-pointer dereferencing vulnerability could also save some user-controlled data on the stack by manually tuning the parameters, which is uncommon in practice. As this finding was not backed up by a principled methodology, it was a "lucky shot." In contrast, we will show that automatic control of the uninitialized memory can be achieved in a more general way.

B. Contributions

In this work, we show that we can meet requirement R2 without an additional memory-corruption vulnerability or special assumptions. In particular, we show that almost the whole kernel stack is controllable to a *local attacker* by either executing syscalls based on how they leave the stack after they return or exhausting memory and guiding stack allocation. We first survey existing reported and patched uninitialized-use vulnerabilities in the Linux kernel and then propose the reliable targeted stack-spraying technique to write and retain arbitrary data on the kernel stack.

The core of the fully automated targeted stack spraying includes a deterministic stack spraying technique and a reliable exhaustive memory spraying technique. The deterministic stack spraying technique consists of three components: a *tailored symbolic execution engine* that explores paths and outputs the concrete parameters to trigger the paths; a *guided fuzzer* that takes as input information generated by the symbolic execution engine to verify that stack control is indeed achieved; and a *coordinator* that safely and efficiently parallelizes the symbolic execution engine and the fuzzer. The exhausting memory spraying technique complements deterministic stack spraying by strategically consuming a huge region of memory to guide stack allocations and preparing malicious data in the memory pages that will be used by the guided stack allocations. Combining both approaches allows us to reliably control almost the whole kernel stack.

We have implemented both deterministic stack spraying and exhausting memory spraying. The deterministic stack sprayer is based on the S2E [11] symbolic execution engine and the Trinity [18] syscall fuzzer. As we need concrete test cases to use S2E, we implemented an automated test case generator that produces S2E test cases for each syscall. To maximize the coverage, we also implemented an S2E plugin that identifies loops in the kernel so that our guided fuzzer can selectively explore loop-related parameters. The exhaustive memory sprayer is implemented as a user-level program that runs before triggering an uninitialized-use vulnerability. Using `kprobes` [2], we also implemented a checker that scans the stack memory at each syscall entry or syscall return to verify that we can indeed control the kernel stack.

To evaluate the performance of the targeted stack-spraying, we measured the range, the distribution, and the frequency of control and the time it takes to achieve control. Our evaluation results show that we are able to control an impressive range of 91% of the kernel stack in total. While exhaustive memory spraying reliably controls 89% of the kernel stack on average, deterministic stack spraying controls 32% of the frequently used stack region, which cannot be reached by exhaustive memory spraying. By adapting Cook's attack to our technique, the attacker can automatically prepare the malicious pointer on the kernel stack and successfully launch an arbitrary memory write or even a privilege escalation attack without the need for known memory corruptions or any special assumptions.

In a nutshell, this paper makes the following contributions:

- We propose automated targeted stack-spraying, which reliably writes arbitrary data to the kernel stack.
- We leverage tailored symbolic execution and guided fuzzing to deterministically control the frequently used stack region, and design a strategy to control dynamically allocated kernel memory, including the kernel stack.
- We show that uninitialized memory on the kernel stack is controllable. Future memory-safety techniques should include uninitialized use as a prevention target.
- We propose a practical mitigation against uninitialized-use exploits with negligible performance overhead.

II. Uninitialized Uses and the Kernel Stack

A. *Uninitialized Uses in OS Kernels*

In this section, we present uninitialized-use issues in OS kernels. We first investigate how widespread uninitialized-use vulnerabilities actually are in the Linux kernel and how aware people are of this problem. To this end, we have manually analyzed the reported Common Vulnerabilities and Exposures (CVE) entries that lead to privilege escalation attacks in the Linux kernel since 2004 [36], and the commit log of the Linux kernel git repository [48], which dates back to 2005. To reduce the huge number of commits to a manageable size, we mostly concentrated on the commit log messages between the years 2015 and 2016. For the CVEs, we find that eight out of 199 (4%) privilege escalation vulnerabilities reported since 2004 are caused by the use of uninitialized objects or pointers. For Linux kernel commit messages, we first identified candidates of uninitialized use by inspecting the commit messages using keywords such as *uninitialized pointer dereference* and *undefined pointer*, which resulted in 52 candidate commits from 2015 and 2016, 28 of which were subsequently filtered out by our manual analysis because they are not exploitable (e.g., NULL pointer dereference bugs). Out of the remaining 24 cases, eight are uninitialized pointer-based reads, which can lead to information leaks, and 16 are uninitialized pointer-based writes or function calls, which are particularly interesting to attackers. We further inspected these 16 interesting cases and found that 11 cases (69%) are from the stack while only five cases are from the heap. These findings not only show that uninitialized-use vulnerabilities are quite common in the Linux kernel but also indicate that these vulnerabilities are not considered particularly security-relevant or even not reported at all. Moreover, our findings confirm that most uninitialized variables are from the stack rather than the heap, which is a significant difference from use-after-free vulnerabilities.

B. *Kernel Stack Management*

Since most uninitialized variables are from stack, our primary focus lies on vulnerabilities caused by uninitialized uses of stack variables and pointers in the Linux kernel, and thus understanding how Linux manages its kernel stacks and which features it offers in this regard is important. In Linux, every thread has its own kernel stack allocated in the kernel memory space with the maximum size of the stack depending on the specific Linux version. In general, the stack is 4KB or 8KB for a 32-bit OS (x86) and 8KB or 16KB for a 64-bit OS (x86-64), which is quite small compared to the default stack size soft limit of 8MB for Linux user space stacks. The special data structure `struct thread_info`, whose size is 104-byte in our system, is saved at the stack top (low address). The fundamental goal behind limiting the kernel stack size is to limit overall memory consumption when a large number of threads is running in the kernel in parallel, and each thread has its own kernel stack. Because of the limited stack size, storing large variables on the kernel stack and creating long call chains in the kernel space is discouraged. To ensure that the stack depth is shallow enough to avoid a stack overflow, Linux provides the `checkstack.pl` tool for static analysis of the stack. Although the small size of the Linux kernel stack improves the success rate of a stack-spraying attack, the shallow stack depth (or the lack of loops and recursions) limits the spraying range. Besides normal thread stacks, Linux also has other specialized stack types. For example, while debug stacks are used for hardware (interrupt1) and software (INT3) debug interrupts, interrupt stacks are used for external hardware interrupts or for processing software interrupts. Since these stacks do not accept user-controlled data, we do not take them into account and instead focus on normal per-thread kernel stacks that are used when syscalls are issued.

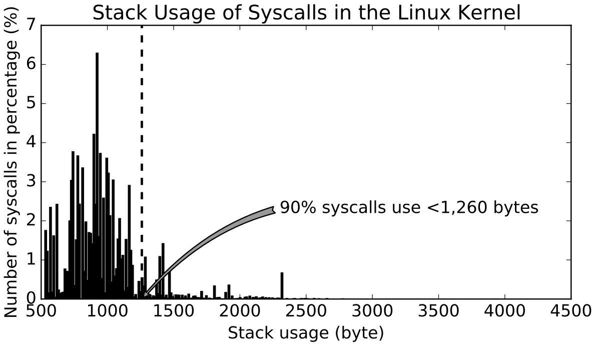

Fig. 2: The profile for stack usage of syscalls in the Linux kernel. The total size of the kernel stack is 16KB. 90% syscalls use less than 1,260 bytes aligned to stack base. The average stack usage is less than 1,000 bytes, and the vast majority of stack objects are allocated within the highest 1KB stack region.

C. *Stack Usage of Syscalls*

The more frequently a stack region is used, the more likely an uninitialized variable will reside in this region. Therefore, taking control over frequently used memory regions increases the success rate of an uninitialized-use attack. We hence analyze stack usage of syscalls to understand which portions of the kernel stack are most frequently used.

To profile stack usage of syscalls, we use `kprobes` to intercept syscall enters and returns, and scan the stack memory to check maximum stack usage of these syscalls. Specifically, upon syscall enter, we zero out the stack memory and continue the normal execution; upon syscall return, we scan the stack memory from stack top (i.e., the lowest stack address) until we find the first non-zero byte. We conservatively treat the offset of the first non-zero byte into stack base (i.e., the value of stack pointer upon syscall entry) as the maximum stack usage of the syscall. We use the `Trinity` fuzzer to invoke all syscalls to obtain stack usage for all syscalls. Because Trinity usually takes a long time to explore a syscall or even just does not terminate, we set five-second timeout for fuzzing each syscall. Figure 2 summarizes the maximum stack usage for all syscalls. In particular, we find that (1) the average stack usage of syscalls is less than 1,000 bytes (aligned to the stack base at high address) and (2) 90% syscalls use only the highest 1,260 bytes on the stack. It is important to note that the stack usage represents the maximum stack region a syscall uses. Assuming stack objects are uniformly distributed in stack regions used by syscalls, we find that the average location of stack objects is 510 bytes into the stack base and more than 90% stack objects are allocated in the highest 960-byte stack region. Therefore, *the highest 1KB stack region is frequently used and thus is the primary target of our spraying*.

III. The Targeted Stack-Spraying Approach

The main challenge in exploiting uninitialized uses is to control data in the uninitialized memory. By planting malicious pointers in the target memory, an uninitialized pointer dereference can be turned into arbitrary memory read/write or code execution. However, unlike heap spraying, in which the number and the size of allocated heap objects are user-controlled, stack spraying has the additional problem of stack objects usually being static and fixed in size. The placement of the Linux `thread_info` structure, at the stack top, requires the stack size to be limited; otherwise, stack buffer overflows may occur. In addition, kernel space is shared by all threads. Not limiting the size of stack will easily exhaust memory. Therefore, Linux kernel developers are encouraged to use the script (`scripts/checkstack.pl`) to statically analyze stack usage. The script in particular checks the stack usage (in byte) of each function so that developers can find functions that use too much stack memory. Because of these features—the limited stack size, the static and fixed-size stack objects, and the stack usage check, a targeted stack-spraying attack is significantly more difficult than a heap-spraying attack.

To enable a targeted stack-spraying attack in the kernel space, we need to prepare malicious data in a specific location of the kernel stack in the presence of aforementioned difficulties. Specifically, the location itself needs to be chosen in such a way that the uninitialized memory will overlap the prepared data. In general, we can store malicious data in such a location in two ways: (1) finding an execution path that prepares data overlapping that of the vulnerability and (2) finding a way to guide the kernel to allocate stacks on the memory pages with prepared data. The first method is deterministic: Once such a path and its triggering inputs are found, we can deterministically write arbitrary data at the specified location. Since the data is saved at the target location by normal execution paths, this method is stealthy and hard to detect. By contrast, the second method affects the stack allocation of another process/thread by exhausting memory, which can be reliable but not fully deterministic. This method can achieve broad control because the overlapping is at page level. However, since the creation of a new process/thread executes kernel functions that use the kernel stack, a portion (near the stack base) of the prepared data will be overwritten. As a result, the second method loses control of the stack region at high address. As mentioned in §II-C, our primary spraying target is the highest 1KB stack region. To control this region, we have to use the first method. For these reasons, we combine both methods so that attackers can achieve reliable or even deterministic control over a broad stack region. In this section, we present an overview of both methods.

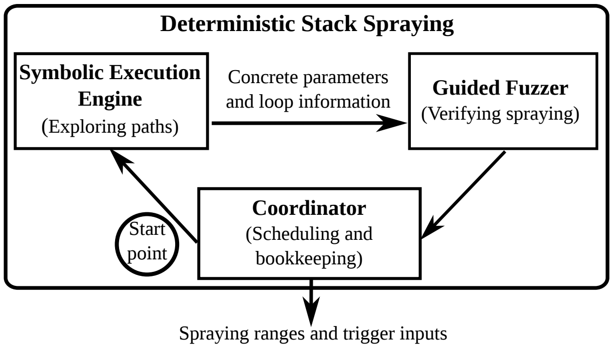
Deterministic Stack Spraying
Symbolic Execution Engine
(Exploring paths)
Concrete parameters and loop information
Guided Fuzzer
(Verifying spraying)
Start point
Coordinator
(Scheduling and bookkeeping)
Spraying ranges and trigger inputs

Fig. 3: Overview of the architecture of our deterministic stack spraying technique that consists of three components. It automatically analyzes all syscalls and outputs results, including which range of the stack we can control and how to control the stack.

A. Deterministic Stack Spraying

We design the deterministic stack spraying technique, which finds an execution path that prepares data overlapping that of an uninitialized variable. The main challenge of deterministic stack spraying is to find a syscall with specific parameters that will trigger an overlapping execution path. An overview of the technique used for the attack is shown in Figure 3. The technique consists of three components: a symbolic execution engine, a guided fuzzer, and a coordinator that handles communication between the symbolic execution engine and the guided fuzzer. The goal of the symbolic execution engine is to explore as many execution paths as possible to find one that saves user-controlled data on that stack, which will overlap an uninitialized variable. However, symbolic execution is prone to the path explosion problem because of that the number of feasible paths in a program can be infinite when the program contains unbounded loop iterations. A possible solution for this problem is to use heuristics for either path-finding or concretizing the loop condition. To achieve high coverage in path exploration, we follow the second method: During symbolic execution, we concretize the loop conditions and at the same time, identify loops and their symbolic conditions, and then let the fuzzer selectively explore these loops. To verify whether a syscall can actually save arbitrary data on the kernel stack, our guided fuzzer replaces the non-controlling parameters (that are confirmed not to affect execution paths during symbolic execution) with a magic code. When the syscall returns, we use `kprobes` to intercept the execution and scan the kernel stack to check which ranges of the stack memory have been polluted by magic code. These ranges are those we can control.

B. Exhaustive Memory Spraying

The exhaustive memory spraying technique guides the stack allocation of a new process or thread so that the memory pages used by the stack overlap those with prepared data. The main challenge of such a technique is to improve the reliability of the overlapping. To overcome this challenge, we design a strategy that reliably controls the memory of the kernel stack. Specifically, our exhaustive memory spraying technique includes two steps: (1) occupying the majority of memory in the target machine and (2) polluting all the available remaining memory with malicious data (for uninitialized variables). Memory occupation forces the kernel to use the remaining memory, which is small, for the newly allocated kernel stacks. Because the remaining memory is small, the pollution operation can be done quickly and effectively. Once we ensure that almost all available memory is polluted by malicious data, the memory of the newly allocated stacks will contain the malicious data. Note that in the kernel space, the kernel does not zero out the allocated memory pages, so the malicious data will not be cleared.

IV. Design

In this section, we discuss design choices we made for both deterministic stack spraying and exhaustive memory spraying.

A. *Deterministic Stack Spraying*

Our primary spraying goal is to deterministically control the frequently used stack region (the highest 1KB stack region), which is likely used by uninitialized variables. To this end, we need to find a suitable syscall and set its parameters such that its execution will write the data in the location, and to verify that the data is retained after the syscall returns. We design the deterministic stack spraying technique, which includes three parts: symbolic execution that explores execution paths, guided fuzzing that verifies spraying, and coordination that safely runs symbolic execution and guided fuzzing in parallel.

1) Symbolic Execution of Syscalls: To find syscalls for deterministic stack control, we need to iterate over possible execution paths of syscalls as completely as possible and generate the concrete parameters that trigger these paths. Since symbolic execution can explore execution paths in a target program and generate concrete inputs to trigger the respective paths, it is an ideal tool for our purpose. For each syscall, we use symbolic execution to iterate over its execution paths and generate concrete inputs that we can then use to verify if an execution path saves data in a target location on the kernel stack. To symbolically execute the Linux kernel, we can adopt two widely used symbolic execution engines, KLEE [7] and S2E [11], both of which are capable of handling C/C++ programs. KLEE is built on top of the LLVM compiler infrastructure while S2E is based on QEMU, which enables S2E to do full-system symbolic execution. Moreover, compared to KLEE, S2E can perform analyses in-vivo within a real software stack (e.g., user programs, libraries, kernel, and drivers) instead of using abstract models of these layers. Even more importantly, S2E supports binaries and employs the selective symbolic execution mechanism to boost performance. Considering these features, we choose S2E as our symbolic execution engine.

Automatic generation of test cases. Since S2E does not automatically decide which variables should be symbolized, it requires as input not only the program to be tested but also a list of variables it should replace with symbolic values. In our case, we have to explicitly tell S2E which buffers, including their address and size, to symbolize. As an example, Figure 4 shows how to symbolically execute the `open` syscall. Using the `s2e_make_symbolic` feature, we explicitly tell S2E to symbolize the `pathname` parameter by specifying the pointer of `pathname` and its size (i.e., `PATH_SIZE`).

```
char pathname[PATH_SIZE];
int flags = O_RDWR;

s2e_enable_forking();
/* symbolize the  pathname parameter */
s2e_make_symbolic(pathname, PATH_SIZE, "pathname");

/* symbolically execute the open syscall */
int res = open(pathname, flags);

s2e_disable_forking();
s2e_kill_state(0, "program terminated");
```

Fig. 4: This example shows how to symbolically execute the `open` syscall in s2E. Here, we symbolize only the `pathname` parameter but not the `flag` parameter. `s2e_enable_forking` is a S2E feature that enables parallel execution upon branches.

Given that current Linux kernel has about 300 syscalls and many of which have up to six parameters, manually writing test cases would be highly time-consuming and therefore impractical. Therefore, we opted for an automatic approach to generate test cases used as input for S2E. However, automatic test case generation entails two challenges: (1) Some syscalls depend on other syscalls and therefore have to be called in a proper order. For example, `read/write` syscalls cannot be called before the `open` syscall; and (2) for pointer-type parameters, we are usually unable to specify the size of the buffer referred to by the pointer. To overcome the first challenge, we rely on the Linux Test Project (LTP) because it properly sets up execution conditions for each syscall. For the second challenge, we observe that execution paths (i.e., control flows of the syscall) are often independent of the number of elements located by pointer-type parameters and thus the size of the respective buffer. Therefore, for symbolic execution, we conservatively assume pointer-type parameters always point to a single element, but later, we will use the guided fuzzer to explore the syscall with more elements (see §IV-A2). Apart from these challenges, the automatic test case generation is intuitive: We generate the C source code that iteratively symbolizes each parameter using the syscall definition with type information of parameters. The syscall definition itself is directly derived from the Linux kernel source code.

Path Exploration. When running the QEMU virtual machine in the S2E mode, executing a test case will automatically trigger symbolic execution. During this phase, each program state represents an execution path. Whenever a state is terminated, i.e., execution of a path is finished, we tell S2E to generate and output sample parameters that trigger this execution path. These sample parameters are then passed to the guided fuzzer for further verification described in §IV-A2. Since the verification process relies on the presence of magic code, which is stored on the stack using syscall parameters, S2E needs to tell the fuzzer which parameters can be replaced by magic code and which need to take on a sample value. The criterion used to distinguish these two types of parameter is their relevance to the control flow of the program: If a parameter is used in a control-flow relevant condition, i.e., it affects the execution path of the program, it is considered a controlling parameter, and thus the fuzzer uses a sample value for it; otherwise, it is considered a non-controlling parameter and can be replaced by magic code. To distinguish controlling and non-controlling parameters, we obtain the path constraints when a state is terminated; if the parameter is used as a constraint, it as a controlling parameter; otherwise, it is a non-controlling parameter.

Identifying Loops. Loops that repeatedly save user-controlled data on the kernel stack are ideal for targeted stack-spraying because they may write arbitrary data to a large region of the stack. Unfortunately, although symbolic execution can help explore execution paths with a high coverage, it generally cannot handle loops properly when the looping condition is also a symbolic value [35]. We address this limitation of symbolic execution by offloading the path exploration for loops to the guided fuzzer. Specifically, we identify the loops during the symbolic execution and provide loop information, (i.e., the looping condition in the form of the respective parameter and its value range to the guided fuzzer). The fuzzer then focuses on exploring this particular parameter in the particular range. However, identifying loops in S2E is challenging. Traditional loop detection mechanisms rely on a dominator tree [22] to extract the dependence relationships among blocks. The dominator tree, however, is not available in S2E because it transforms the binary code of a program into an LLVM IR representation block by block thus losing the information about dependencies among blocks. Without this information, a precise identification of loops is infeasible. Since false positives in identifying potential loops during symbolic execution only introduce more work for the fuzzer, we use a two-layered approach to conservatively identify loops during symbolic execution. The first layer is an execution history-based identification, and the second is based on the relative offsets between instructions. Specifically, in the first layer, when given a function, we maintain the list of executed instructions, and whenever a conditional jump is executed, we check its target: If it targets an already executed instruction, we identify it as a loop. This execution history-based approach, however, is unable to detect a loop if it is executed only once. In this case, we invoke the second layer of our approach to further check the address of the jump target: If the address is lower than the one of the conditional jump instruction, we also identify it as a loop. It is important to note that the relative offset-based check is also not entirely reliable since it is possible that a conditional jump for a loop may target a higher address, resulting in false negatives. Nonetheless, our two-layered approach works reasonably well and can largely solve the loop identification problem in a timely manner. Once we have successfully identified a conditional jump that is used for looping, we extract the loop condition from the comparison instruction. By checking whether the loop condition is a symbolic value, we are able to determine whether the loop condition is dependent on the syscall parameters. To further reduce the search space for the guided fuzzer, we also query the constraint solver of S2E for the possible value ranges of the symbolic loop condition. These value ranges are then used to guide the fuzzing process.

2) Guided Fuzzing: The fuzzing mechanism verifies that the targeted stack-spraying is indeed achieved when executing the kernel with the inputs generated by symbolic execution. As shown in Figure 3, the guided fuzzing mechanism takes as input the output of the symbolic execution component (i.e., the sample parameters for the respective syscall and, if present, additional loop information). The result of the fuzzing phase is essentially an overview of what we have to do in order to control the kernel stack, including which syscalls, which parameters of these syscalls we need to use, and the effect of these syscalls on the kernel stack (i.e., which range of the stack we can control).

Verifying spraying. To verify whether spraying is achieved (i.e., the magic code is left on the kernel stack), first, the guided fuzzing prepares the concrete parameters for the syscalls reported by the symbolic execution component, which are either the sample parameters generated by S2E or magic code. As described in §IV-A1, we assume a pointer-type parameter always points to a single-element buffer. To reduce crashes caused by out-of-bound accesses and increase the spraying range during the fuzzing, for a pointer-type parameter, we `mmap` a memory of the size of the kernel stack, fill in it with magic code, and let the pointer-type parameter point to this memory. Second, we need to scan the stack memory right after a syscall returns. Therefore, we need to intercept the return of the syscall and dump the stack memory at the point of the return. The methods to intercepting syscalls include: (1) instrumenting the Linux kernel source code, (2) patching the syscall table, and (3) using kprobes [2]. Method (1) might introduce a bias when verifying the success of the targeted stack-spraying because it requires changing the source code. Therefore, this method is not desirable. Method (2) and method (3) are similar in principle; however, since kprobes provides a more flexible and reliable way of intercepting syscalls, we chose method (3) to intercept the return of syscalls and insert our logic in the handler for the interception. Upon intercepting a syscall return, the verification is performed by scanning the stack memory and checking which ranges of stack memory have been polluted with the magic code. Once the magic code is found, the range information together with the corresponding syscall parameters are reported.

Fuzzing loops. A well-known limitation with symbolic execution is the path explosion problem that the number of feasible paths in a program grows exponentially in the case of programs with loops. Even a single loop can generate a huge number of symbolic execution paths corresponding to the loop iterations [35], thus resulting in the path explosion problem. To handle this problem, KLEE (internally adopted by S2E) randomly picks or uses search heuristics [7] to select a state to execute. This design decision inherently prevents our targeted stack-spraying from finding and exploiting a syscall containing a loop to spray a huge and continuous range of stack. Therefore, instead of letting S2E symbolically execute the loop, we let it tell us which syscall contains loops and which parameters are used as the loop condition. Then we let our guided fuzzer handle loop cases by specifically fuzzing the condition-related parameters. All other parameters that are used as input for the fuzzer are either the sample values generated by S2E or magic code, as mentioned in §IV-A1. With the combination of S2E (with the loop information) and the guided fuzzer, we are able to efficiently and comprehensively identify the controllable range of the stack.

3) Coordination: The coordination unit is designed to safely run the symbolic execution and the guided fuzzing components in parallel. Both the symbolic execution engine and the guided fuzzer are contained in a QEMU virtual machine, for which two separate QEMU snapshots are created that specifically run either the symbolic execution engine or fuzzing mechanism. Instead of running all syscalls in the same instance of a snapshot, each syscall is tested in a separate instance, thereby enabling us to run the syscalls independently of each other and thus safely contain the crash or the error of the running of each syscall. To facilitate the coordination between the different components, we have designed two features: (1) a communication scheme for the whole testing framework and (2) a controlling scheme for sending commands to the testing instances.

Communication scheme. To efficiently find the syscall and its parameters that can achieve the targeted stack-spraying, the symbolic execution snapshot and the guided fuzzing snapshot run in parallel; therefore, a real-time communication scheme is required. The communication is used for sending (1) commands from the coordinator to the virtual machines, (2) outputs of the symbolic execution engine to the guided fuzzing mechanism, and (3) verification results from the guided fuzzing mechanism to the coordinator. We chose `pipe` as the communication channel and use the paravirtualized drivers (`virtio`) [3] of KVM to improve the performance of I/O operations.

Controlling scheme. During the analysis process, it is common that the symbolic execution engine and the guided fuzzing mechanism crash or get stuck in infinite loops. In these cases, the controlling scheme must terminate or restart the virtual machines. Specifically, we design a command receiver along with a set of pre-defined commands, which runs in both snapshots of the QEMU virtual machine for symbolic execution and guided fuzzing. As an example, when the guided fuzzing does not terminate after a specified period of time, the coordinator will send the command `STOP` to terminate the snapshot.

B. Exhaustive Memory Spraying

Although the deterministic stack spraying technique can deterministically control the frequently used stack region, its coverage is limited: It is hard to find an execution path that can save attacker-controlled data in the stack region after the highest 1KB because stack objects are rarely saved in this region. To control this region that spans the majority of the kernel stack, we design the exhaustive memory spraying technique. Note that this technique is general: It can control not only the kernel stack but also other memory regions that are dynamically allocated in the kernel space, such as the kernel heap. Compared to the deterministic stack spraying technique, the exhaustive memory spraying technique is straightforward, which includes two parts: (1) memory occupation, which consumes the majority of available memory in a system and (2) memory pollution, which writes malicious data in the remaining memory.

1) Occupying Memory: The goal of occupying the majority of available memory is to restrict the kernel to use the small remaining memory for new stack allocations. Because the remaining memory is small, attackers can finish the next step, memory pollution, in a quick and effective manner. To decide how much memory to occupy, we first obtain the total size of available memory and allow attackers to specify the amount of non-occupied memory (e.g. 50MB). All other available memory is then to be occupied. Specifically, we incrementally create many processes, each of which `mmaps` a few megabytes of memory. To avoid being "shrunk" by techniques `Copy-on-Write` and `Deduplication`, we explicitly write a random 8-byte value (obtained from `/dev/urandom`) into each memory page. Moreover, we keep these processes running during the attack to ensure the memory remains occupied throughout the exhaustive memory spraying process.

2) Polluting Memory: Since the majority of memory has already been occupied, when the kernel creates a new process or thread, the allocated kernel stack will be forced to use the remaining available memory. By polluting the remaining memory with malicious data (that attackers want to spray into the kernel stack), the allocated kernel stack will overlap the memory pages with the malicious data. The polluting process also obtains the size of available memory (after memory occupation), `mmaps` a memory of this size and writes the malicious data into it. Afterwards, the pollution process `munmaps` the polluted memory. Note that `munmap` does not clear the malicious data in memory. To ensure that the pollution is effective and that the polluted memory contents are not overwritten by another process, we perform the `munmap` operation right before invoking the syscall with an uninitialized-use vulnerability.

V. Implementation

In this section, we present the prototype of both the deterministic stack spraying technique and the deterministic stack spraying technique technique. Although the implementation currently targets the Linux kernel, it is possible to extend it to other OS kernels (e.g. windows) since it only requires the syscall interfaces and can directly work on binaries.

A. Deterministic Stack Sprayer

1) Symbolic Execution Engine: As discussed in §IV-A1, we use S2E as the symbolic execution engine in our targeted stack-spraying system. To facilitate the analysis of large numbers of syscalls, we additionally implemented an automatic test case generator for syscalls and two S2E plugins that handle the input generation for the fuzzer and the identification of loops.

Test case generator. The automatic test case generator takes as input the definition (i.e., the signatures) of the syscalls to be tested. To obtain these definitions, we searched the source code of the Linux kernel for the pattern of syscall definition. Specifically, syscalls are always defined using the uniform macro `SYSCALL_DEFINEx` where `x` denotes the number of parameters. For example, the `open` syscall is defined as follows:

```
SYSCALL_DEFINE3(open, const char *, filename, int, flags, uint16_t, mode)
```

Given that we achieve targeted stack-spraying by preparing data in parameters, syscalls that do not have parameters (e.g., `getpid`) and therefore cannot take user-controlled data are ignored. Also, because the underlying hardware architecture of our testing machine is x86/x86_64, only syscalls for this architecture are selected. The test case generator itself is implemented as a python script with the execution environment being set up by LTP. As mentioned in §IV, when handling pointer-type parameters, we only symbolize the first element of the buffer the pointer refers to. To reduce potential issues caused by out-of-bound accesses, during symbolic execution, we allocate memory chunks of the kernel-stack size (i.e., 16KB) for these elements. Once the respective parameters are symbolized, we use the general `syscall()` function to trigger the symbolic execution for the syscall under analysis.

S2E Plugins. We implemented two S2E plugins to facilitate the automatic analysis of large numbers of syscalls: (1) a path explorer plugin that explores possible execution paths and generates concrete parameters for each execution path, and (2) a loop explorer plugin that identifies loops whose looping conditions depend on syscall parameters. The path explorer plugin intercepts the state-killing signal that occurs when an execution path is finished, i.e., when a state is terminated. The signal handler then asks the constraint solver to generate sample parameters that trigger this path. For each parameter, the plugin further checks if it is contained in the path constraints: If yes, the parameter is classified as a controlling parameter that affects the execution path; otherwise, it is a non-controlling parameter that will be replaced with magic code during the guided fuzzing. The loop explorer plugin aims to identify loops with symbolic conditions. We identify loops using a two-layered approach that tracks the execution history for each function as well as the relative offsets between these instructions inside the respective function. Specifically, we hook S2E at the end of each block by catching the `onTranslateBlockEnd` signal and then check if the last instruction of the block is a direct call or indirect call instruction to intercept function calls. Note that S2E translates the binary at block level rather than function level, so checking the last instruction of each block is necessary to identify function calls. When the execution enters a function, we create a list to maintain the executed instructions. Since the (virtual) address of each instruction in the memory is unique, we save the PC values (i.e., the values of instruction pointer) in the list. To know whether an instruction is a conditional jump, we read the machine code pointed to by the PC to get the opcode and then match the opcode to confirm the instruction. Once a conditional jump is identified, the PC of the next instruction (i.e., the target of the jump) is checked: If it is already in the executed instruction list, we report it as a loop. Otherwise, we check whether its PC is smaller than the one of the conditional jump instruction: If it is, we also report it as a loop. If the loop condition is symbolic, we use the `getRange()` function provided by the constraint solver to compute the possible value range of the condition value.

2) Guided Fuzzer: The guided fuzzer verifies if the spraying can be actually achieved and also mitigates the limitations of symbolic execution by specifically fuzzing loop-related parameters. Our guided fuzzer is implemented on top of the Trinity fuzzer [18].

Tailoring the Trinity fuzzer. Fuzzing in general faces the problem that purely randomized inputs for functions often lead to failures (i.e., being terminated by sanity checks), preventing the exploration of interesting execution paths. For example, if a file descriptor parameter (4-byte) would be purely randomized, the kernel would likely simply reject the execution of the syscall and return `-EINVAL`. Trinity addresses this problem by creating a list of file descriptors: opening pipes, scanning file systems (e.g., `sysfs`, `procfs`, and `/dev`), and creating a bunch of sockets using random network protocols; and then passing one of these entities at random whenever a syscall needs a file descriptor. For parameters other than file descriptors, we instruct Trinity to take as input the concrete parameters generated by the symbolic execution component. In particular,if a parameter is loop-related (i.e., it is used as the looping condition), we let Trinity focus on this parameter by generating random values within the value range specified by the symbolic execution component.

Spraying verifier. As discussed in §IV-A2, intercepting syscall returns is done by using the `kprobes` tool. `kprobes` provides three different types of probes depending on the intended purpose: kprobe for intercepting syscall entries, jprobe for intercepting jump instructions, and kretprobe for intercepting syscall entries and returns. Since we check the stack at syscall returns, the most suitable probe for us is kretprobe, which we implemented in a kernel module. Our kretprobe kernel module takes as input the name of syscall under analysis and intercepts the return of this particular syscall. Upon interception, we derive the stack top (the lowest address) of the current kernel stack by computing `stack_pointer & (THREAD_SIZE - 1)` where `stack_pointer` is provided by `kprobes` and the macro `THREAD_SIZE` defines the size of the kernel stack, which varies on operating systems, e.g. it is typically 8K on 64-bit Ubuntu and 16KB on 64-bit Debian. Once we have computed the stack top and size of the kernel stack, verification is performed by searching the stack memory for magic code. To output the verification results, we need to write the data into user space files from our kernel module. For safety and reliability reasons, writing to a user space file from kernel space is discouraged. Therefore, we use the `/proc` virtual file system to pass results to user space, which is later passed to the coordinator.

3) Coordinator: The coordinator controls the symbolic execution engine and the guided fuzzer. The input for the coordinator is the specific range of stack that we want to control, which is used to tell our targeted stack-spraying to find syscalls with corresponding parameters that can save arbitrary data in this range. Once it receives that range, the coordinator runs the symbolic execution engine and the guided fuzzer in parallel to identify suitable syscalls and their parameters as quickly as possible.

Ranking and syscalls. To identify syscalls that can spray the specified range as quickly as possible, the coordinator prioritizes three types of syscalls: (1) syscalls that set configurations or write data. Such syscalls (e.g., `pwritev`) are very likely to save data on kernel stack; (2) syscalls that contain loops. Such syscalls usually affect a large region of the kernel stack; and (3) syscalls that contain (multiple) pointer-type parameters. By directing pointers to attacker-controlled buffers containing magic code, syscalls containing more pointer-type parameters are likely to have a higher chance to save the magic code on kernel stack. Remaining syscalls are analyzed after the aforementioned ones. With more interesting syscalls being analyzed first, it is more likely to find a suitable sprayer in a fixed amount of time.

B. Exhaustive Memory Sprayer

We implemented the exhaustive memory sprayer as a user-space program. The amount of available memory in the system is obtained with command `free -m`. In our case, we want to keep some small portion of memory (e.g. 50MB) available and occupy all other memory. During memory occupation, we fork processes, each of which occupies 2MB memory, to exhaust all memory besides that we want to intentionally leave available.

The polluting process then writes malicious data (magic code in our case) to the remaining available memory and munmaps it. To verify if the newly allocated stacks use the polluted memory pages, we intercept syscall entries using kprobes and scan stack memory for magic code. We instrumented the Trinity fuzzer to asynchronously run memory occupation and call memory pollution before invoking syscalls.

VI. Evaluation

We evaluated the effectiveness of our targeted stack-spraying approach with regard to exploiting uninitialized-use vulnerabilities by measuring the control coverage we achieved. We present the total stack ranges that we can control with deterministic stack spraying and exhaustive memory spraying, the distribution of controlled regions, and the time spraying takes. In particular, we investigate the following questions:

- **Stack spraying coverage.** What is the overall range of the kernel stack can we control with our two spraying techniques?
- **Coverage distribution and frequency.** In deterministic stack spraying, how is the control coverage distributed over the kernel stack? And how frequently can we control a specific stack region?
- **Spraying reliability.** In exhaustive memory spraying, how reliably can we control memory?
- **Spraying efficiency.** How long does it take for our spraying techniques to achieve memory control?

A. Experimental Setup

We obtained the symbolic execution engine S2E from the master branch of its git repository[1], which uses QEMU 1.0.50 and clang 3.2. Our guided fuzzer is based on Trinity version 1.7pre. Both the symbolic execution and guided fuzzer run on virtual machines with Debian 8.5.0 (64-bit) on Linux kernel version 3.16. We selected syscalls in the way described in §V; out of 313 syscalls available in the kernel source, we selected 229 for the evaluation. The stack of the Debian system is 16K-byte. The stack has two regions that are at fixed locations and cannot be sprayed: the lowest 104 bytes reserved for thread_info and the highest 160 bytes reserved for OS operations such as context switches. In all evaluations, the magic code is set to be 4-byte string "UBIE".

B. Stack Spraying Coverage

We evaluated the coverage for deterministic stack spraying and exhaustive memory spraying separately and then measured their combined coverage. In both scenarios, we used 229 pre-selected syscalls for the evaluation.

In deterministic stack spraying, we found that only 34 syscalls do not allow us to take control of any stack region. After manual inspection, we concluded that this is because these syscalls do not admit any parameters that will be stored on the stack. Table I summarizes the amount of bytes that can be controlled by the top 10 syscalls. In the highest 1KB stack region, which is frequently used (§II-C), deterministic stack spraying covers 315 bytes using all available syscalls. Hence, 32% of the frequently used region of the kernel stack can be manipulated using deterministic stack spraying.

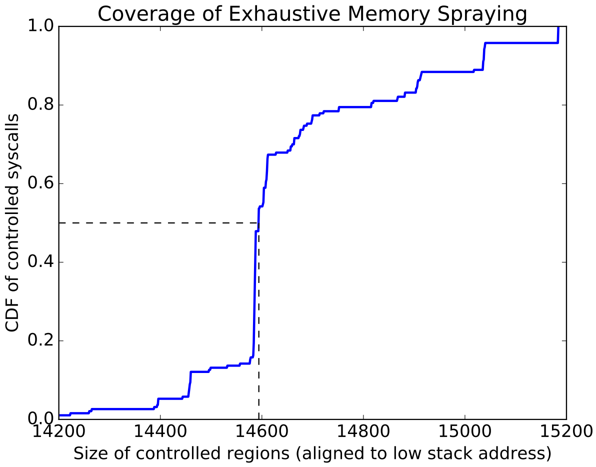

Fig. 5: The cumulative distribution (CDF) of coverage achieved by exhaustive memory spraying. Its average control rate is about 90%. The controlled memory is aligned to the low address of the kernel stack; a portion (1,700 bytes on average) near the stack base cannot be controlled.

Stack memory after the highest 1KB is subjected to exhaustive memory spraying. As mentioned in §III, a portion of the prepared malicious data in the kernel stack of a victim process by exhaustive memory spraying is likely to be overwritten because of some kernel operations (e.g., setting up a new process) in the victim process. To evaluate which areas are overwritten, we enabled exhaustive memory spraying and ran the Trinity fuzzer to invoke syscalls. We then used kprobes to intercept syscall entry points and check which regions have been polluted by magic code (indicating that they were successfully sprayed). Figure 5 shows the results: Besides a small overwritten region near stack base, the remaining region can be fully controlled. The size of the uncontrollable region varies. On average, the highest 1,722 bytes at the stack base are overwritten, and in some cases, this region can be as small as 1,200 bytes. Overall, while losing control of this region, exhaustive memory spraying retains control of all other stack memory, achieving an average coverage rate of 89%.

Deterministic stack spraying and exhaustive memory spraying work as two complementary techniques: While exhaustive memory spraying retains the majority of the memory, it cannot control the frequently used stack region. Deterministic stack spraying complements it by controlling 32% memory of the frequently used stack region. Overall, by combining both spraying techniques, targeted stack-spraying reliably controls more than 91% of the kernel stack.

C. Coverage Distribution and Frequency

We further investigated how the control coverage is distributed over the kernel stack when using deterministic stack spraying. Figure 6 presents the distribution. We found that the coverage ranges from 200 to 800 bytes. More importantly, the control with deterministic stack spraying is highly concentrated: Some regions can be controlled by more than 100 syscalls. We

[1] https://github.com/dslab-epfl/s2e.git as of August 2016

System call	Coverage (Byte)
vmsplice	224
uname	99
fcntl	96
setpriority	88
sched_get_priority_min	88
sched_get_priority_max	88
personality	84
iopl	84
umask	80
io_destroy	76

TABLE I: Top 10 syscalls with highest individual control coverage in the kernel stack.

System call	Unique Coverage
wait4	16
waitid	12
timerfd_create	8
clock_getres	8
fcntl	8
mq_open	8
sched_rr_get_interval	8
mq_notify	8
timer_gettime	4
Total	**80**

TABLE II: Syscalls that uniquely control a stack region. Unique coverage is the number of uniquely controlled bytes.

believe these regions are most likely used by stack objects, and uninitialized variables likely reside in these regions, so controlling these regions is critical to exploit uninitialized uses from the kernel stack. Table I presents top 10 syscalls with high coverage. Syscalls `vmsplice`, `uname`, and `fcntl` have the highest individual coverage. We further investigated which regions of the stack are uniquely controlled by a syscall. Table II contains all syscalls that uniquely control a region. Overall, only 80 bytes are uniquely controllable by a single syscall. Other covered bytes can be controlled with multiple syscalls, thus their sprayers are more reliable.

D. Reliability of Exhaustive Memory Spraying

We investigated the reliability of exhaustive memory spraying by measuring how likely the kernel uses the sprayed memory for the kernel stack, i.e. whether the allocated stack memory overlaps the one with prepared data. Specifically, we enable the exhaustive memory spraying and run the `Trinity` fuzzer to invoke syscalls. Then we count the number of times (i.e., probes) a syscall has been invoked until we find that the kernel stack for the syscall is sprayed. After running all syscalls with `Trinity`, we found that in most cases, the kernel uses the sprayed memory as stack in the first or second probe. The average number of probes we achieve overlapping is 1.8. The worse case is less than 10 probes. Such results show that the exhaustive memory spraying technique is very effective and thus can reliably control the uninitialized memory.

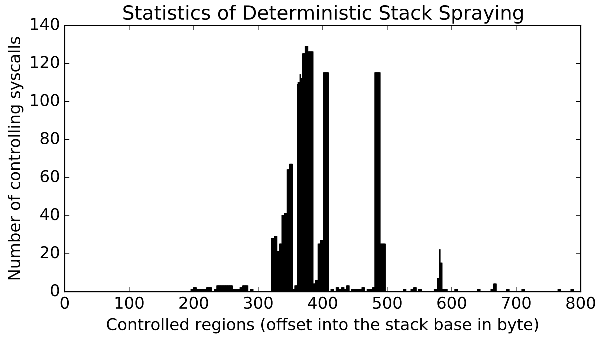

Fig. 6: The coverage, distribution, and frequency of stack control achieved by the deterministic stack spraying technique.

E. Efficiency of Spraying

In deterministic stack spraying, both the symbolic execution and the guided fuzzing are time-consuming processes. In many cases, they do not terminate even after running for a few hours. To handle this problem, we have set a timeout for this analysis: If the analysis for a syscall does not generate new coverage within a pre-set timeout, we forcibly terminate the analysis for this syscall and continue to analyze next one. After experimenting with various timeouts, we ultimately set the timeout to 30 minutes. We found that the vast majority of syscalls can be thoroughly analyzed within this time-frame, with only 12 syscalls not finishing in time. With the syscall ranking mentioned in §V-A3, we were able to control more than 200 bytes in the frequently used region within a few minutes. Compared to deterministic stack spraying, exhaustive memory spraying is much more efficient. The time memory occupying takes depends on the size of the available memory. In our case, the memory is 512MB, and the time for occupying the memory is less than 2 seconds. Since memory pollution writes data into a small memory region, its time is unobservable.

F. Case Study

The targeted stack-spraying technique provides a conceptual approach for exploiting a given uninitialized-use vulnerabilities by preparing malicious data at a target stack location. For the sake of illustration, we exemplify the applicability of our approach by adapting Cook's exploit [12].

To the best of our knowledge, Cook's exploit is the only one that exploited an uninitialized-use vulnerability (CVE-2010-2963) in the Linux kernel stack. Figure 7 shows how the code is subject to the uninitialized use. The pointer `data` in object `karg.vc` is not initialized but dereferenced in function `copy_from_user()`. Cook exploited this vulnerability by tuning the `cmd` argument to let the union struct adopt the type of `struct video_tuner`, causing `karg.vt` to be written with user-controlled data. For such a spraying attack to succeed, at least four requirements must be satisfied: (1) The object having the uninitialized pointer must be contained by a union struct; (2) another type in the union struct has to have a non-pointer field that overlaps with the uninitialized pointer because users are not allowed to specify pointers pointing to kernel space; 3) this non-pointer field can be overwritten with user-controlled data; and 4) the user-controlled data will not be cleared. An execution path satisfying all these requirements is uncommon in practice, and finding such a path manually is unrealistic in most cases.

```
static long do_video_ioctl(struct file *file, unsigned int cmd,
        unsigned long arg) {
    union {
        struct video_tuner vt;
        struct video_code vc;
    } karg;
    ...
    /* karg.vc contains an uninitialized pointer */
    err = get_microcode32(&karg.vc, up);
    ...
}
int get_microcode32(struct video_code *kp,
        struct video_code32 __user *up) {
    ...
    /* uninitialized pointer is dereferenced */
    copy_from_user(kp->data, up->data, up->datasize))
    ...
}
```

Fig. 7: The uninitialized-use vulnerability used in Cook's exploit.

In this case study, we show that our targeted stack-spraying technique can automatically find many execution paths that are able to prepare a malicious pointer on the kernel stack, thus controlling the uninitialized pointer `kp->data`. To reproduce Cook's exploit, we installed version `2.6.27.58` of the Linux kernel in 64-bit Ubuntu 14.04; the kernel source code in file `compat_ioctl32.c` was reverted to contain the vulnerability described in CVE-2010-2963. Determined by the operating system, the size of the kernel stack is 8KB instead of 16KB in this case study. As mentioned in §IV, to benefit from our targeted stack-spraying technique, we need to find out the location of the uninitialized pointer in the stack. To get the pointer location, we used `kprobes` to hook the function `do_video_ioctl`. The handler provided by `kprobes` enables us to find the location of the stack pointer when `do_video_ioctl` is called. Using this information, we computed the offset of the uninitialized pointer `kp->data` from the stack base, which is `396`. After knowing this offset, we employed our deterministic stack spraying technique to find syscalls that can prepare a 8-byte malicious pointer at this offset. Altogether, we were able to find 27 such syscalls with corresponding parameters. Independent of the chosen syscall, we could always prepare a malicious pointer at the target offset, resulting in an arbitrary write.

This case study shows how to use the proposed deterministic stack spraying technique to find syscalls that can control a specific location on stack. It also confirms that control of the stack can be achieved generally and automatically, and, in the presence of a suitable uninitialized-use vulnerability, a successful exploit can be built reliably and readily.

VII. Mitigation

We showed that uninitialized-use vulnerabilities can be readily and reliably exploited using our targeted stack-spraying technique. While use-after-free and buffer overflow problems have been extensively studied, which has resulted in various protection techniques (e.g., memory safety), the uninitialized-use problem has rarely received attention. Our findings show that uninitialized use constitutes a severe attack vector that calls for practical defense mechanisms; however, to date no such defense mechanisms exist. As such, we designed and implemented an efficient and practical mitigation that counters uninitialized uses. Our mitigation is inspired by the observation that uninitialized-use exploits usually control an uninitialized pointer to achieve arbitrary read/write/execution. By zero-initializing pointer-type fields in an allocated object, we can prevent an adversary from controlling these pointers. Since memory page at the address zero is not accessable in Linux[2], zero-initialization becomes a safe prevention operation. More specifically, we perform an intra-procedural analysis for the Linux kernel source code. We realize both the analysis that identifies unsafe pointer-type fields and the instrumentation that zero-initializes the identified pointer-type fields based on the LLVM compiler Infrastructure [24].

A. *Identifying Unsafe Pointer-Type Fields*

Our analysis is carried out on the LLVM IR, so type information is preserved. In most cases, we can differentiate pointer-type fields from other fields based on type information. We start our analysis from allocation sites (i.e., `AllocaInst` instructions in LLVM IR). The first step is to identify all pointer-type fields by recursively (a field could be a struct or an array type) traversing each field in the allocated object. Since integer values might be used as a pointer, we also treat the 8-byte-integer type as a pointer type.

To initialize identified pointer-type fields, we could conservatively zero out them upon allocations. This strategy, however, will overly initialize many already initialized pointers and therefore introduce unnecessary performance overhead. To reduce initialization overhead while still ensuring security, we designed an intra-procedural program analysis that checks the following two conditions: (1) the pointer field is not properly initialized in the function; and (2) the pointer is sinking (e.g., being used or passed to other functions). Only those pointer-type fields satisfying both conditions require zero-initialization. More specifically, once all pointer-type fields are identified, we perform taint analysis to keep track of the initialization status and sinking status of the pointer-type fields in the following conservative ways:

- When a pointer-type field is explicitly assigned by other values (i.e., it is the store-to target in a memory storing instruction (`StoreInst`)), we record that this field is initialized.
- When a pointer-type field that is not fully initialized is passed to other functions as a parameter or stored to memory, we report it as unsafe, which thus requires initialization.
- When a pointer-type field that is not fully initialized is dereferenced (i.e., used as the pointer argument in memory loading instruction (`LoadInst`), `StoreInst`, or function call instruction (`CallInst`), we treat it as unsafe as well.

The basic alias analysis [23] provided by LLVM is adopted to tackle the alias problem, so accessing pointer-type fields via their aliases is also tracked. Since our analysis is intra-procedural, such a basic alias analysis suffices for the purpose of efficiently detecting pointer-type fields that lack proper initialization. With this conservative taint analysis, we managed to reduce the number of to-be-initialized bytes from 105,960 to 66,846.

[2] Since the Linux kernel with version 2.6.23, the `/proc/sys/vm/mmap_min_addr` tunable was introduced to prevent unprivileged users from creating new memory mappings below the minimum address

System call	Baseline	W/ defense	Overhead(%)
null syscall	0.04	0.04	(0.0%)
stat	0.42	0.40	(-4.8%)
open/close	1.20	1.14	(-5.0%)
select TCP	2.44	2.62	(7.4%)
signal install	0.11	0.11	(0.0%)
signal handle	0.60	0.64	(6.7%)
fork+exit	163	157	(-3.7%)
fork+exec	447	460	(2.9%)
prot fault	0.327	0.356	(8.9%)
pipe	8.906	9.058	(1.7%)
TCP	25.6	27.5	(7.4%)
Average			**1.95%**

TABLE III: LMBench results. Time is in microsecond.

B. Implementation

Both the analysis pass and the instrumentation pass are implemented with LLVM. Both passes are inserted after all optimization passes. To use the mitigation, users only need to specify the option (i.e., `-fsanitize=init-pointer`) when compiling the kernel source code. To compile Linux kernel source code with LLVM, we applied the patches provided by the LLVMLinux project [25]. The zero-initialization code is inserted right after allocation sites. In LLVM IR, inline assembly is invoked by a `CallInst`, which is treated as a sink in our analysis, so the common inline assembly in Linux kernel is not an issue.

C. Evaluating Pointer Initialization

To confirm that our mitigation is practical, we applied it to the latest Linux vanilla kernel (x86_64, version `4.7`) and evaluated its performance. The testing is performed in the virtual machine with the secured kernel. The host machine is equipped with an Intel(R) Core(TM) i7-2760QM CPU @ 2.40GHz processor and 10GB of RAM; the running OS is 64-bit Ubuntu 14.04 with Linux kernel version `3.13.0-55`. The virtual machine (VirtualBox) was configured to have a 4-core processor and 4GB RAM; its OS is also 64-bit Ubuntu 14.04. We used the default configuration to compile the kernel code.

Performance with system services. We used LMbench [27] as the micro benchmark to test the runtime slowdown in system services. The selected system services are mainly syscalls, which conform to typical kernel performance evaluations (e.g., [19]). The evaluation results are shown in Table III. The average performance overhead is only 2%, and in most cases, the performance overhead is less than 5%. These numbers confirm that our zero-initialization-based mitigation for kernel stack is efficient.

Performance with user programs. We further used the SPEC CPU 2006 benchmarks as a macro-benchmark to test the performance impacts of our mitigation over the user-space programs. We ran the test 10 times and adopted the average number. Table IV shows the evaluation results. Our zero-initialization-based mitigation imposes almost no performance overhead (0.47%) to the SPEC benchmarks on average.

Both the LMbench and SPEC benchmark results confirm that our mitigation is very efficient and reliable (no single error was observed during the evaluation).

Programs	Baseline	W/ defense	Overhead(%)
perlbench	3.62	3.62	(0.0%)
bzip2	4.74	4.75	(0.2%)
gcc	0.945	0.945	(0.0%)
mcf	2.71	2.68	(-1.1%)
gobmk	13.9	13.9	(0.0%)
hmmer	2.02	2.03	(0.5%)
sjeng	3.28	3.30	(0.6%)
libquantum	0.0365	0.0365	(0.0%)
h264ref	9.35	9.40	(0.5%)
omnetpp	0.342	0.349	(2.0%)
astar	7.77	7.74	(-0.4%)
xalancbmk	0.0611	0.0611	(0.0%)
milc	4.47	4.51	(0.9%)
namd	8.84	8.85	(0.1%)
dealII	10.5	10.6	(1.0%)
soplex	0.0201	0.0201	(0.0%)
povray	0.407	0.417	(2.5%)
lbm	1.66	1.68	(1.2%)
sphinx	1.16	1.17	(0.9%)
Average			**0.47%**

TABLE IV: User space (x86_64) performance evaluation results with the SPEC benchmarks. Time is in second, the smaller the better.

VIII. Related Work

In this section, we provide a compact overview of the offensive and defensive related works.

A. Memory Spraying

Memory spraying is a popular means to memory-corruption attacks. By far the most popular memory spraying techniques is *heap spraying*, an attack that was first described by SkyLined in 2004 [38]. Heap spraying attacks fill large portions of the victim's heap memory with malicious code (e.g., NOP sleds), thus increasing the chance of hitting malicious code for hijacking the control flow [14, 15]. Although the heap spraying technique itself has been countered by the introduction of Data Execution Prevention (DEP), the evolution of heap spraying—*JIT spraying*—has become a popular concept for enabling a variety of web-based attacks [42]. JIT spraying exploits the predictability of the JIT compiler to create predictable code fragments that can be used to hijack control-flow [42, 50]. Since these fragments reside in an executable code cache, mitigation techniques like DEP or $W \oplus X$ can be bypassed [42, 50]. Existing defenses against heap/JIT spraying attacks either try to detect the attack by searching for large amounts of NOP sleds and shell code [14, 15, 38] or randomizes the memory layout and register assignments [13, 14, 50]. Recently, memory spraying has also been used to exploit the "Rowhammer" vulnerability in DRAM devices where repeated access to a certain row of memory causes bit flips in adjacent memory rows [5, 40].

In contrast to all these existing spraying techniques, our targeted stack-spraying target the stack instead of the heap. More importantly, our stack spraying technique is deterministic and stealthy (thus is hard to detect), and our exhaustive memory spraying technique is highly reliable.

B. Kernel Exploits and Automated Exploits

Since the kernel is often a part of the trusted computing base of a system, avoiding exploitable kernel vulnerabilities is critical for the security of a system [9]. Nonetheless, despite the efforts of kernel developers to find and eliminate these vulnerabilities, new such vulnerabilities are still frequently detected. As of the paper writing, a total of 1,526 vulnerabilities have been reported in the Linux kernel alone, 203 of which were reported in 2016 [37]. With Linux kernel vulnerabilities being on the rise, corresponding exploitation techniques have caught the interests of attackers. One recent approach exploits use-after-free vulnerabilities in the Linux kernel by leveraging its memory recycling mechanism [51], while another one circumvents existing defenses by manipulating the kernel page table [21].

Although many vulnerabilities and their corresponding exploits are still discovered manually, automatic detection and exploit generation is becoming increasingly popular, as is evidenced by the DARPA Cyber Grand Challenge (DARPA CGC) [43]. In this challenge, teams are required to build automated vulnerability scanning engines, which they then use to compete in a Capture The Flag tournament. One of the tools specifically developed for this challenge is Fuzzbomb [28], which combines static analysis with symbolic execution and fuzzing to detect vulnerabilities in programs. The combination of symbolic execution and fuzzing is also used for the Driller tool [45], which has also been tested on 126 of the DARPA CGC binaries. Driller, like our approach, uses symbolic execution to guide its fuzzing engine in case it fails to generate input to satisfy complex checks in the code. This combination is also used together with static and dynamic program analysis to automatically generate exploits for a wide variety of applications [47]. Similar to these approaches, we also use a combination of symbolic execution and fuzzing to discover execution paths that can achieve targeted spraying in the Linux kernel.

C. Uninitialized Use Exploits

Despite the fact that uninitialized-use bugs are seldom considered to be security-critical, a number of exploits for these vulnerabilities have become known in recent years. Flake [16] used a manual approach towards exploiting uninitialized local variables on the user-space stack, while Cook [12] used an unchecked `copy_from_user()` call with an uninitialized variable to exploit the Linux kernel and gain root privileges. Jurczyk in turn exploited CVE-2011-2018, a stack-based uninitialized-variable reference vulnerability in the Windows kernel, which allows an attacker to execute arbitrary code with system privileges [17]. Last but not least, Chen exploited an heap-based uninitialized-use vulnerability in Microsoft's Internet Explorer (CVE-2015-1745) using fuzzing [8]. Unlike these ad-hoc attacks, our targeted stack-spraying is general and automated.

D. Uninitialized Use Detection and Prevention

Researchers have proposed some detection mechanisms for uninitialized uses; however, only few defenses against uninitialized uses have been proposed. For detection, tools such as kmemcheck [33], Dr.Memory [6], and Valgrind [41] leverage dynamic instrumentation and analysis to track memory accesses while compiler-based approaches like MemorySanitizer [44] and Usher [52] insert tracking code to find uninitialized uses at runtime. For defense mechanisms, Kurmus and Zippel [20] proposed a technique for preventing exploits of memory-corruption vulnerabilities. Their approach relies on single-split kernels where system calls of untrusted processes can only access a hardened kernel version while trusted processes can access the unmodified kernel. A solution that is specifically targeted towards uninitialized-use vulnerabilities is offered by the PaX team, known for the invention of ASLR. Their GCC compiler plugins, STACKLEAK and STRUCTLEAK, clear the kernel stack on kernel-to-user transitions and initialize all local variables that might be copied to user space, which effectively prevents uninitialized uses of kernel memory [46]. A key difference of our efficient defense against uninitialized uses is that instead of initializing all local variables, we specifically initialize pointer-type fields that have not been properly initialized before. While STACKLEAK and Split kernel introduce a significant performance overhead (e.g., STACKLEAK introduces an average of 40% runtime overhead in system operations [26]), our lightweight defense imposes almost no performance overhead.

E. Memory Safety Techniques

Memory-corruption errors such as dangling pointers are a long-known problem in unsafe programming languages like C. In the last ten years, several approaches have been proposed to mitigate the exploits of these errors.

Watchdog [32] and its successor WatchdogLite [29] both leverage hardware supports to store and check allocation meta-data to prevent use-after-free vulnerabilities. Softbound [30] and CETS [31] are software-based approaches that aim to prevent memory-corruption errors at compile-time. Softbound enforces spatial memory safety by storing base and bound information as metadata for every pointer, while CETS enforces temporal memory safety by storing unique identifiers for each object, which are then used to check if the object is still allocated upon pointer dereferences. Notably, although these memory safety techniques claim full memory safety, they currently do not cover uninitialized use as a prevention target. In contrast to these metadata-based approaches, DieHard [4] and its successor, DieHarder [34] both focus on randomizing the location of heap objects to make dangling-pointer dereferences hard to exploit. Since both techniques focus on heap objects, they cannot detect and prevent uninitialized-use errors on the stack. StackArmor [10] also adopts randomization to achieve the memory safety for stack. All these randomization-based memory-safety techniques are probabilistic.

IX. Discussion

In this section, we discuss the potential limitations of targeted stack-spraying and corresponding defenses. We also discuss the requirements to port targeted stack-spraying to other programs such as web browsers.

A. Exploitability of Uninitialized-Use Vulnerabilities

Not all uninitialized-use vulnerabilities are exploitable. First, in order to benefit from targeted stack-spraying, the execution path that triggers an uninitialized-use vulnerability must not overwrite the prepared malicious data. Otherwise, the targeted stack-spraying technique will lose control of the uninitialized memory thus cannot exploit this uninitialized-use vulnerability.

To verify if the prepared data persists until triggering the uninitialized-use vulnerability, attackers can obtain the address of the instruction using the uninitialized memory and use `kprobes` to intercept the instruction to verify if the prepared data persists. Second, our current deterministic stack spraying does not consider the case in which the preparation of the malicious data occurs in the same syscall that also triggers the uninitialized-use vulnerability. We ensure only that the prepared malicious data persists until the entry point of the syscall triggering the uninitialized-use vulnerability.

B. Porting to Other Programs

To port the deterministic stack spraying technique to other programs such as the JavaScript engine in web browsers, we need interface definition (e.g. the JavaScript API), a targeted symbolic execution engine (e.g Kudzu [39]), and a fuzzer (e.g. jsfunfuzz [1]). Test suites are usually available for well-maintained programs, which can be used to automatically generate the test cases needed for symbolic execution and fuzzing. When these resources are available, the deterministic stack spraying technique can be conveniently ported to support other programs. To port the exhaustive spraying technique to other programs, we only need to provide the function for allocating large memory and the size of available memory.

C. Improving Mitigation and Other Defenses

As mentioned in §VII, we can efficiently mitigate uninitialized-use exploits by zero-initializing all pointer-type fields for which the compiler cannot prove that they are properly initialized before reaching a sink (i.e., they are used). This lightweight approach works for most cases. However, false negatives cannot be fully excluded: If a pointer is modified by (or depends on) an uninitialized non-pointer value, zero-initializing this pointer cannot effectively prevent the exploits because the resulting pointer is still controllable by attackers if they can control the non-pointer value. Therefore, one possible improvement for our proposed defense is to zero-initialize non-pointer values as well. Two approaches that already offer a broader defense in this respect are PaX's STACKLEAK [46] and split kernel [20] (see §VIII for details). Both approaches provide strong security to prevent uninitialized-use exploits, but come at the cost of a significant runtime overhead [26]. As such, a sophisticated inter-procedural and field-sensitive analysis is necessary to filter out safe allocations. We leave this challenging problem for future work.

Another defense direction is to defeat targeted stack-spraying. A mitigation against the deterministic stack spraying technique is to randomly adjust the stack base upon syscall entry so that the malicious data prepared in the previous syscall may not overlap the uninitialized variable in the vulnerable syscall. Since the kernel stack has only 8KB or 16KB, the entropy of such a randomization is limited. To detect the exhaustive memory spraying technique, systems can monitor a large amount of process creations or large memory allocations. However, this spraying technique can be stealthy by reducing the amount of process creations and the size of memory allocations, and probing more times.

X. Conclusion

Using uninitialized variables (uninitialized use) constitutes a common type of memory error in the Linux kernel. Reliably exploiting uninitialized uses on the kernel stack has been considered infeasible since the code executed prior to triggering the vulnerability must leave an attacker-controlled pattern on the stack. As a consequence, uninitialized uses are widely tolerated as undefined behaviors, and full memory safety techniques such as SoftBound+CETS therefore exclude uninitialized use as a prevention target. Moreover, uninitialized uses are even intentionally used as a randomness source by popular systems such as OpenSSL.

We have shown in this paper that uninitialized use constitutes a severe attack vector that future memory safety techniques should seriously defend against. We have proposed the fully automated targeted stack-spraying approach, which includes a deterministic stack spraying technique and an exhaustive memory spraying technique. While exhaustive memory spraying reliably controls 89% of the kernel stack on average, deterministic stack spraying controls 32% of the frequently used stack region, which cannot be reached by exhaustive memory spraying. Therefore, attackers can use the targeted stack-spraying approach to readily exploit an uninitialized-use vulnerability for a privilege escalation attack. To mitigate uninitialized use exploits, we have proposed a compiler-based mechanism, which initializes potentially unsafe pointer-type fields with almost no performance overhead.

Acknowledgment

We thank Chengyu Song, Taesoo Kim, Insu Yun, and the anonymous reviewers for their valuable feedback. This work was supported by the German Federal Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA). Kangjie Lu and Wenke Lee were supported in part by the NSF award CNS-1017265, CNS-0831300, CNS-1149051, CNS-1563848 and DGE-1500084, by the ONR under grant N000140911042 and N000141512162, by the DHS under contract N66001-12-C-0133, by the United States Air Force under contract FA8650-10-C-7025, by the DARPA Transparent Computing program under contract DARPA-15-15-TC-FP-006, by the ETRI MSIP/IITP[B0101-15-0644]. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the BMBF, NSF, ONR, DHS, United States Air Force, DARPA or MSIP.

References

[1] jsfunfuzz: a JavaScript-based fuzzer, 2016. https://github.com/MozillaSecurity/funfuzz.

[2] Kernel Probes, 2016. https://www.kernel.org/doc/Documentation/kprobes.txt.

[3] Virtio: Paravirtualized drivers for kvm/Linux, 2016. http://www.linux-kvm.org/page/Virtio.

[4] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[5] E. Bosman, K. Razavi, H. Bos, , and C. Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, USA, May 2016. IEEE.

[6] D. Bruening and Q. Zhao. Practical memory checking with dr. memory. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, Washington, DC, Mar. 2011.

[7] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[8] Chen. Hey Man, Have You Forgotten To Initialize Your Memory?, 2015. URL https://www.blackhat.com/docs/eu-15/materials/eu-15-Chen-Hey-Man-Have-You-Forgotten-To-Initialize-Your-Memory.pdf.

[9] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

[10] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[11] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[12] K. Cook. Kernel Exploitation Via Uninitialized Stack. 2011. https://www.defcon.org/images/defcon-19/dc-19-presentations/Cook/DEFCON-19-Cook-Kernel-Exploitation.pdf.

[13] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780. IEEE, 2015.

[14] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 327–336. ACM, 2010.

[15] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106. Springer, 2009.

[16] H. Flake. Attacks on Uninitialized Local Variables. 2006. http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf.

[17] M. "j00ru" Jurczyk. The story of CVE-2011-2018 exploitation, 2012. URL http://j00ru.vexillium.org/blog/20_05_12/cve_2011_2018.pdf. [Online; accessed 16-Aug-2016].

[18] D. Jones. Trinity: A Linux System call fuzz tester, 2015. http://codemonkey.org.uk/projects/trinity.

[19] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[20] A. Kurmus and R. Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1366–1377. ACM, 2014.

[21] J. Lee, H. Ham, I. Kim, and J. Song. Poster: Page table manipulation attack. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1644–1646. ACM, 2015.

[22] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1), Jan. 1979.

[23] LLVM. LLVM Alias Analysis Infrastructure, 2016. http://llvm.org/docs/AliasAnalysis.html.

[24] LLVM. The LLVM Compiler Infrastructure, 2016. http://llvm.org/.

[25] LLVMLinux. The LLVMLinux Project, 2016. http://llvm.linuxfoundation.org/index.php/Main_Page.

[26] K. Lu, C. Song, T. Kim, and W. Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[27] L. W. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.

[28] D. J. Musliner, S. E. Friedman, M. Boldt, J. Benton, M. Schuchard, P. Keller, and S. McCamant. Fuzzbomb: Autonomous cyber vulnerability detection and repair. In *Fourth International Conference on Communications, Computation, Networks and Technologies (INNOV 2015)*, 2015.

[29] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of the 2014 International Symposium on Code Generation and Optimization (CGO)*.

[30] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[31] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management*, 2010.

[32] S. Nagarakatte, M. M. Martin, and S. Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 189–200. IEEE Computer Society, 2012.

[33] V. Nossum. Getting Started With kmemcheck, 2015. https://www.kernel.org/doc/Documentation/kmemcheck.txt.

[34] G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[35] J. Obdrzalek and M. Trtik. Efficient loop navigation for symbolic execution. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*, ATVA'11, 2011.

[36] S. Özkan. CVE Details: Linux kernel security vulnerabilities - gain privilege, 2016. URL https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/opgpriv-1/Linux-Linux-Kernel.html. [Online; accessed 12-Aug-2016].

[37] S. Özkan. CVE Details: Linux kernel security vulnerabilities - overview, 2016. URL https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33l. [Online; accessed 15-Aug-2016].

[38] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[39] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[40] M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 2015.

[41] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, June–July 2005.

[42] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *NDSS*, 2015.

[43] J. Song and J. Alves-Foss. The darpa cyber grand challenge: A competitor's perspective. *IEEE Security & Privacy*, 13(6):72–76, 2015.

[44] E. Stepanov and K. Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO)*.

[45] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[46] P. Team. PaX - gcc plugins galore, 2015. URL https://pax.grsecurity.net/docs/PaXTeam-H2HC13-PaX-gcc-plugins.pdft. [Online; accessed 15-Aug-2016].

[47] H. A. Thanassis, C. S. Kil, and B. David. Aeg: Automatic exploit generation. In *ser. Network and Distributed System Security Symposium*, 2011.

[48] L. Torvalds. Linux Kernel Git Repository, 2016. URL git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git. [Online; accessed 5-Aug-2016].

[49] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*, Seoul, South Korea, July 2012.

[50] T. Wei, T. Wang, L. Duan, and J. Luo. Secure dynamic code generation against spraying. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 738–740. ACM, 2010.

[51] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425. ACM, 2015.

[52] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *Proceedings of the 2014 International Symposium on Code Generation and Optimization (CGO)*.