I. Introduction

Modern operating systems such as NetBSD, macOS, and Windows isolate their kernel from user space programs to increase fault tolerance and to protect against malicious manipulations [10]. User space programs have to call into the kernel to request resources, via system calls or ioctls. This communication between user space and kernel space crosses a security boundary.

Kernel memory disclosures - also known as kernel information leaks - denote the inadvertent copying of uninitialized bytes from kernel space to user space. Such disclosed memory may contain cryptographic keys, information about the kernel memory layout, or other forms of secret data. Even though kernel memory disclosures do not allow direct exploitation of a system, they lay the ground for it.

We introduce KLEAK, a simple approach to dynamically detect kernel information leaks. Simply said, KLEAK utilizes a rudimentary form of taint tracking: it taints kernel memory with marker values, lets the data travel through the kernel and scans the buffers exchanged between the kernel and the user space for these marker values. By using compiler instrumentation and rotating the markers at regular intervals, KLEAK significantly reduces the number of false positives, and is able to yield relevant results with little effort.

Our approach is practically feasible as we prove with an implementation for the NetBSD kernel. A small performance penalty is introduced, but the system remains usable. In addition to implementing KLEAK in the NetBSD kernel, we applied our approach to FreeBSD 11.2. In total, we detected 21 previously unknown kernel memory disclosures in NetBSD-current and FreeBSD 11.2, which were fixed subsequently. As a follow-up, the projects’ developers manually audited related kernel areas and identified dozens of other kernel memory disclosures.

The remainder of this paper is structured as follows. Section II discusses the bug class of kernel memory disclosures. Section III presents KLEAK to dynamically detect instances of this bug class. Section IV discusses the results of applying KLEAK to NetBSD-current and FreeBSD 11.2. Section V reviews prior research. Finally, Section VI concludes this paper.

II. Kernel Memory Disclosures via Uninitialized Memory

The main reason leading to memory disclosures is uninitialized memory that the kernel inadvertently copies to userland. According to Lu et al. [9], uninitialized memory was the most common
cause for kernel memory disclosures in the Linux kernel during the period between January 2010 and March 2011. The root cause are peculiarities of the C programming language and its compilers, such as data structure alignment. Since memory disclosures are not functional bugs developers often overlook them. Unfortunately, compilers do not help developers to mitigate this problem in case of complex structures and unions. Please refer to [7] for an in-depth discussion of the root causes that lead to kernel memory disclosures.

Memory disclosures can be exploited by attackers under certain conditions. For instance, the kernel stack is typically reused between system calls. Suppose there are two system calls syscall\textsubscript{target} and syscall\textsubscript{vulnerable}, where the first stores secret information on the kernel stack and the latter discloses several bytes from the kernel stack. By calling syscall\textsubscript{target} just before syscall\textsubscript{vulnerable}, an attacker may be able to read the secret kernel stack content.

Techniques like Linux’s STACKLEAK [1] try to mitigate this by erasing the kernel stack after each system call. They do not, however, prevent the leaks from occurring. In addition, they introduce a performance penalty that is not acceptable in a production system.

III. KLEAK

This section presents the approach of KLEAK to dynamically detect kernel memory disclosures. In a nutshell, KLEAK utilizes a simple form of dynamic taint tracking [13] to detect such leaks. It introduces taint at memory sources like malloc in the form of 1-byte-sized marker values (Section i). It then scans outgoing buffers at the kernel-userland frontier for the aforementioned marker bytes (Section ii). Particular marker values are chosen to reduce false positives (Section iii). The values are then rotated in several rounds in order to further reduce false positives (Section iv). We finally discuss the implementation of KLEAK in the NetBSD kernel (Section v) and the limitations of our approach (Section vi).

i. Tainting Memory Sources

KLEAK taints two types of memory sources: the kernel stack and dynamically-allocated kernel buffers. While the kernel stack serves for local variables with a short life time, which is less than the duration of a system call, dynamically-allocated kernel buffers may outlive the life time of a system call. Regardless of the type, KLEAK taints it with a 1-byte-sized marker, whose value varies over time.

Dynamically-allocated Memory

The first type of memory sources is dynamically-allocated memory. We instrument dynamic memory allocation functions, i.e. malloc and kmem_alloc, to return marked memory chunks. In other words, we add memset calls, and fill the areas with a repeated pattern of the marker values. One exception are requests that require memory set to zero. These requests are not marked in order to ensure the functionality of the kernel.

Kernel Stack

The second type of memory source is the kernel stack. We taint the kernel stack before each system call. This is done at the kernel system call entry point, where we call a function that allocates an array on the kernel stack and initializes it with the marker value. This array is big enough to cover most of the non-consumed part of the stack. Therefore, from then on, the stack can be considered as tainted.
However, a problem arises during execution: the kernel will push and pop values on the stack, operations that will overwrite the taint. We overcome this problem by using the `fsanitize-coverage=trace-pc` compiler instrumentation: on each edge the compiler will insert calls to a specific `_sanitizer_cov_trace_pc` function, in which KLEAK will re-taint a smaller part of the stack. In other words, the kernel is instrumented to systematically taint its stack while it executes. Using this simple method, we are able to significantly increase the probability that an uninitialized byte on the stack will be tainted.

Another similar problem arises on CPU architectures like x86 64bit that use nested stacks while calling Interrupt Service Routines (ISR): if an interrupt is received while a kernel function executes, an ISR will be pushed and will execute on the current kernel stack, which means that the taint will be overwritten. The aforementioned compiler instrumentation eliminates part of the problem, because after the ISR finished executing, it is likely that the previously executing kernel function will reach an edge not too distant in terms of instructions, and this edge will cause a re-taint of part of the overwritten stack. In order to fully address the problem, however, it should be possible, on x86 64bit, to configure the CPU to use the IST (Interrupt-Stack Table) mechanism, and have the ISRs execute on a separate stack. Achieving this involves re-writing the low-level kernel interrupt paths, and it is therefore out of scope of KLEAK.

### ii. Detecting Leaks at the Data Sinks

The data exchanges between the kernel and the user space are performed via a set of specific functions, such as `copyout` on NetBSD or `copy_to_user` on Linux. Therefore, we define such functions as our data sinks. On NetBSD these are `copyout` and `copyoutstr`. On each invocation of these data sinks, we scan the kernel buffers passed to these functions, on a byte-by-byte basis, looking for the marker value.

### iii. Choice of the Marker Values

As mentioned above, the marker is 1-byte-sized. Statistically, the marker will necessarily appear at some point in a legitimate, non-leaky kernel buffer. The choice of the marker value is therefore important, because an uncommon marker will generate fewer false positives than a common one. For instance, the zero byte would be a suboptimal choice since this byte is quite common, typically as a string terminator or initialization byte of `malloc`. Intuitively, the choice of byte 181 as marker value is preferable since it is a prime number that does not have an ASCII representation.

To the best of our knowledge, there was no estimation of byte frequencies that we could have leveraged to choose our marker values. Therefore, we estimated the byte frequencies for `copyout` and `copyoutstr` on NetBSD, by patching `copyout` and `copyoutstr` so that these functions would count the byte values on each invocation. We utilized the NetBSD test suite `ATF` [6] to provoke many invocations of `copyout,copyoutstr`.

Figure [1] shows the results for `copyout` (left) and `copyoutstr` (right). We logarithmically scaled the data. In total, we recorded 116.591.537 calls to `copyout` that transferred 14.093.269.240 bytes and 3.623.846 calls to `copyoutstr` that transferred 83.773.615 non-zero bytes. Note that each call to `copyoutstr` ended with a NUL control character to terminate it. We omit these in the following.
The distribution of bytes that were transferred by copyoutstr unsurprisingly reflects the ASCII range. It transferred printable characters (bytes 32 until 126) as well as the control characters TAB and LF (bytes 9 and 10). In the case of copyout, there is no clear pattern as in the case of copyoutstr. The bytes 0 and 255 are among the top bytes because they are used to express the values zero and minus one. Furthermore, very small values (less than 20) have a high occurrence probability, probably because they represent data sizes or enumeration values. Another observation is that characters of the ASCII spectrum exhibit a high occurrence probability. There is a similar pattern when compared to the plot of the copyoutstr byte distribution. Table 1 summarizes the most seen and least seen bytes.

<table>
<thead>
<tr>
<th>rank</th>
<th>byte</th>
<th>ASCII</th>
<th>rank</th>
<th>byte</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>.</td>
<td>247</td>
<td>207</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>97</td>
<td>a</td>
<td>248</td>
<td>181</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>255</td>
<td>.</td>
<td>249</td>
<td>206</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>101</td>
<td>e</td>
<td>250</td>
<td>167</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td>99</td>
<td>c</td>
<td>251</td>
<td>169</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>d</td>
<td>252</td>
<td>159</td>
<td>.</td>
</tr>
<tr>
<td>7</td>
<td>98</td>
<td>b</td>
<td>253</td>
<td>218</td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td>91</td>
<td>[</td>
<td>254</td>
<td>221</td>
<td>.</td>
</tr>
<tr>
<td>9</td>
<td>234</td>
<td>.</td>
<td>255</td>
<td>157</td>
<td>.</td>
</tr>
<tr>
<td>10</td>
<td>102</td>
<td>f</td>
<td>256</td>
<td>154</td>
<td>.</td>
</tr>
</tbody>
</table>

Table 1: The ten most seen and the ten least seen bytes in copyout during the run of ATF. Note that a dot (.) means that there is no ASCII representation of the byte.

We conclude that marker values should be drawn from the range between byte 140 and 230. Bytes from this range exhibit a low probability of occurrence when compared to the other bytes. Interestingly, some of the least seen bytes are prime numbers (157, 167, 181). In contrast, bytes from the ASCII range as well as bytes at the low and the high limits are suboptimal choices.

iv. Rotation of the Marker Values

In the previous section we established a list of the least probable bytes, usable as markers. Using only one marker value, however uncommon it is, would still produce a considerable number of false positives. We address this problem by allowing the marker to be updated.
An efficient detection method consists in invoking a kernel entry point in several rounds, rotating the marker byte between each round, and considering as leaky only the buffers that were found marked during each of the rounds. The C pseudo-code in Listing 1 illustrates the principle. Statistically, the more rounds we perform, the more we eliminate false positives. From experience we note that performing four rounds is enough to significantly reduce the number of false positives, down to only a few.

```
for (i = 0; i < n_rounds; i++) {
    invoke_kernel_entry_point();
    if (i < n_rounds - 1)
        update_marker();
}
dump_results();
```

Listing 1: Pseudo code of the rotation algorithm of KLEAK.

We maintain a hitmap in the kernel, which is basically a buffer that has the size of the kernel .text section. Within each byte of the hitmap, there are eight bits, each one corresponding to a round. When a data sink is found to be copying a marked byte, we determine the offset of the calling function within the .text section, and set to one the -th bit of the offset-th byte in the hitmap, being the number of the current round. After the last round has been reached, the results are contained in the hitmap: we consider as leaky each copyout that received marked bytes during each of the rounds. For instance, if four rounds were programmed, and one copyout was given marked bytes during each of the four rounds, then the four least significant bits of the associated byte in the hitmap will be set, and we consider this a positive result.

This design implies that only up to eight rounds can be programmed, because there are only eight bits used to register the hits. However, as noted before, four rounds are in practice already sufficient.

v. Implementation

The implementation of KLEAK for the NetBSD kernel comprises less than 600 lines of C code [14].

The marker values as well as the number of rounds are configurable via sysctl. Per default KLEAK is not enabled in the generic NetBSD configuration; we consider KLEAK a developer option, which is not to be used in production kernels.

A user tool is provided, that allows to perform the detection method described in Section iv. It takes as argument a command to execute with KLEAK detection. Listing 2 gives an example of use.

```
$ kleak -n 4 ps
[... output of ps ...]
Possible info leak: [len=1056, leaked=931]
#0 0xffffffff80bad351 in kleak_copyout
#1 0xffffffff80b2cf64 in uvm_swap_stats.part.1
#2 0xffffffff80b2d38d in uvm_swap_stats
#3 0xffffffff80b2d43c in sys_swapctl
#4 0xffffffff80259b82 in syscall
```

Listing 2: Output of KLEAK for a kernel memory disclosure in system call swapctl. Launch the program ‘ps’ with KLEAK enabled, performing four rounds.
vi. Limitations

This section discusses several limitations of our approach.

Simplicity and Speed over Precision

We consciously sacrifice precision to achieve simplicity and speed. Since we do not track the tainted data while it travels through the kernel space, we lose precision. However, this makes KLEAK’s implementation much simpler and faster, when compared to similar approaches like [7]. Even though this may be a limitation, from our point of view this is an advantage: having a simple and fast approach that detects a fair share of kernel memory disclosures.

Code Coverage

Just like every dynamic analysis technique, KLEAK suffers from the fact that it may not reach 100% code coverage. There may be leaky code branches that are seldom executed. Therefore, our system may not detect these kernel memory disclosures. Note that there are ways to increase the code coverage like coverage-guided fuzzing (e.g. syzkaller [3]). Thus, it is possible to push the code coverage further towards 100%.

Portability

The current implementation of KLEAK depends on system specific functionality (e.g. the sysctl parameters, see Section v). While porting it to other BSD-based systems is easy - as shown with our prototype on FreeBSD 11.2 -, porting KLEAK to other UNIX-like systems like Linux may be more time consuming. We believe that this is still feasible thanks to the small code base of our approach. However, it is not possible to independently implement KLEAK in proprietary kernels such as the Windows one, because source access is required.

IV. Results

This section discusses the results of applying KLEAK to NetBSD-current and FreeBSD 11.2.

We targeted the amd64 architecture in both cases. In total, we found 21 kernel memory disclosures. Table 2 summarizes our findings on NetBSD-current and FreeBSD 11.2 as of time of writing. Most of the detected kernel memory disclosures are stack-based: a structure is allocated on the stack, but not properly initialized. Only a few of the memory disclosures are related to dynamically-allocated memory.

The kernel memory disclosure by system call getdirentries of FreeBSD 11.2 showcases how KLEAK complements manual code auditing. We reported a four byte leak of this system call, which was caused by the virtual filesystem operations of UFS. Subsequently, FreeBSD developers audited the code of this system call and its individual implementations in several filesystems; as a result, they found and fixed 17 more kernel memory disclosures in other filesystem implementations.

Similarly, on NetBSD, a memory disclosure initially detected by KLEAK in the amd64 code turned out to be present in the code of several other architectures (such as mips and hppa), leading to 14+ leaks being found and fixed.
Table 2: Summary of the detected kernel memory disclosures in NetBSD-current and FreeBSD 11.2. We ran KLEAK on the AMD64 architecture.

V. Prior Research on Kernel Memory Disclosures

Throughout the years, several security researchers have found kernel memory disclosures in various operating systems via manual code audits and other non-disclosed techniques (e.g. [11, 2]).

Several static analysis approaches focus on the detection of memory disclosures. Compiler toolchains offer little help (e.g. Clang Static Analyzer [8] or gcc’s -Wuninitialized [5] flag) because they approximate the data flow and only work on function scope. Peiro et al. [12] utilize Coccinelle to detect kernel stack memory disclosures in the Linux kernel. Lu et al. [9] present UniSAN. Their static analysis approach checks for memory allocations that leave the kernel if they are fully initialized along all possible execution paths. These unsafe allocations are reported and require an analyst to confirm them as true kernel memory disclosures. The PaX team published several patches for the Linux kernel to mitigate kernel memory disclosures. The STACKLEAK patch, which was recently added to the Linux Kernel by Popov [1] mitigates kernel stack disclosures. STACKLEAK erases the kernel stack before returning from a system call. Note that this mitigation only protects the stack of system calls previous to a vulnerable system call but not the stack of a vulnerable system call itself. Another patch called STRUCTLEAK enforces the initialization to zero of structures that are potentially copied to the user space. It has less performance impact than STACKLEAK but also less coverage.

There are also dynamic analysis approaches. Jurczyk presents the first systematic study on this topic. He utilized full system emulation and precise dynamic taint tracking to discover more than 80 kernel memory disclosures in Windows and Linux [7]. However, this approach suffers from a huge performance penalty due to the full system emulation. anxiaocao and pjf of IceSword Lab (Qihoo 360) [4] utilize a similar approach to Jurczyk but without taint tracking, losing therefore precision.
VI. Conclusion

We presented KLEAK, a methodology that can dynamically detect kernel memory disclosures. First, it taints kernel memory sources with marker values, including the kernel stack and dynamic areas created via malloc. Subsequently, it monitors data sinks (e.g., copyout) by scanning the outgoing buffers for the aforementioned markers. We motivated our choice of marker values in an estimation of least probable bytes. We improved the detection by using compiler instrumentation for the stack, and reduced the number of false positives by rotating the marker value.

As of time of writing, KLEAK is fully implemented in the NetBSD kernel. We evaluated KLEAK on NetBSD-current and FreeBSD 11.2. KLEAK detected 21 kernel memory disclosures that the two projects subsequently fixed. This research is ongoing, and more fixes are to be expected.

VII. Acknowledgements

We would like to thank Martin Clauss for valuable feedback.

REFERENCES


