POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

MemTrace: a dynamic memory overlaps tracing tool

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Kristopher Francesco Pellizzi

Advisor: Prof. Mario Polino

Co-advisors: Prof. Michele Carminati, Prof. Stefano Zanero

Academic year: 2020-2021

1. Introduction

Nowadays mitigation techniques against memory corruption vulnerabilities are commonly implemented in essentially all the major operating systems and compilers. Techniques such as *stack canaries*, $W \oplus X$ and *ASLR* effectively raised the bar, thus increasing the effort needed to write an exploit that allows to inject and execute arbitrary code, as it usually requires a **leak** that allows the attacker to get either the stored *canary* or the address of some memory section or library function.

A frequently used technique to get a leak consists in exploiting **uninitialized memory reads**. Since memory is a limited resource, it is continuously allocated and deallocated during a program's execution according to its needs. This way, memory locations can be reused multiple times, thus possibly generating some **unintended overlaps**. The combination of memory overlaps and uninitialized memory reads may therefore allow to obtain a leak that may be used to bypass mitigation techniques.

The most usual way of approaching a program looking for some leaks is by manually performing a combination of **static** and **dynamic** analysis using tools like *decompilers* and *debuggers*. But such kind of analysis may require a large amount of time. Even with simple programs, it may require **hours** to analyze the possible memory overlaps to try and obtain a leak; while the analysis of more complex programs may also require **days** or **weeks** of analysis.

Some of the existing tools ([6]) allow to analyze a binary looking for possible uninitialized reads. They, however, provide no insight information about what we can read from them, thus leaving the responsibility to perform additional manual analysis to the user. Other approaches to try and detect uninitialized reads or leaks require the availability of the source code or to lift the binary to an *intermediate representation* in order to perform static analysis ([2]), or they make use of *symbolic execution*([3]).

For what concerns the static analysis approach, we must consider 2 main situations:

Source code required: closed-source software is always released without making the source code publicly available, thus preventing the analysis.

Binary lifted to IR: static analysis performed on lifted binaries may generate many **false positives** or **false negatives** due to the lack of semantic information about symbols and variables which are lost during compilation.

The symbolic execution approach, instead, can

be applied to binaries directly. However, symbolic execution is known to be subjected to *path explosion*, which may slow down or even prevent analysis with higher program complexity. Moreover, in order to avoid execution to get stuck, symbolic execution engines usually use *models* of the most frequent and complex library functions. Therefore, if the model over simplifies the actual function, the symbolic execution may generate many **false positives** or **false negatives**, and it can still get the analysis stuck if it is not implemented at all.
We developed *MemTrace*, a new tool which makes use of *Dynamic Binary Instrumentation* (DBI) to detect **uninitialized reads** in a binary and report the memory overlaps that are generated during its execution. We then paired our tool with a well-known fuzzer (*AFL++*) in order to try and explore as much execution paths as possible and report all the overlaps that may happen during binary's execution. Since it is even possible that a binary changes its behavior according to the arguments it is executed with, we leveraged the fuzzer in order to try and perform **command-line arguments fuzzing** as well.
Finally, we tested our tool with a set of binaries having known vulnerabilities. In most cases, the tool, paired with the fuzzer, was able to automatically report the known vulnerability. In all the cases, however, the tool was able to report the vulnerability, when it was triggered by a manually crafted input.
In summary, our main contributions are the following:

- We leveraged DBI to perform a new kind of dynamic binary analysis, which aims at reporting memory overlaps, i.e., uninitialized reads together with write accesses overlapping the same memory location
- We used a fuzzer to explore a program's control flow graph and increase branch coverage
- We leveraged the fuzzer to perform also command-line arguments fuzzing

2. MemTrace

2.1. Overview

The main idea behind *MemTrace* is to execute the binary and keep track of all the executed memory accesses. This way, as soon as it detects an uninitialized memory read, *MemTrace* can report it and look backward at which memory writes were executed last on the same memory location.
As a means to keep track of the *state* of bytes in memory (i.e., either **initialized** or **uninitialized**), *MemTrace* uses a **shadow memory** which associates a bit to each byte of memory used by the analyzed application.
During a program's execution, it is not rare that some values are copied into other registers or other memory locations. So, in order to deal with these *data transfers*, we also designed a **taint analysis** that allows *MemTrace* to keep track of copies of *uninitialized bytes* and therefore detect also *indirect* uninitialized reads and usages of uninitialized bytes.

2.2. Approach & Implementation

MemTrace has been implemented as a dynamic binary analysis tool using *Intel PIN* [4] as the underlying *DBI* framework and makes use of DBI in order to analyze the instructions executed by a binary and detect the ones performing a *memory access*. Whenever a memory access is detected, *MemTrace* will store information about it, so that it will be possible to review the whole history of executed memory accesses. By doing this, *MemTrace* is able to group together *read memory accesses* with all the *write memory accesses* that overlap the same memory location and are not completely overwritten before the execution of the read access itself.
In order to achieve its goal, *MemTrace* must be able to recognize read accesses that read uninitialized data. To do that, *MemTrace* must know, at any moment, the state of memory bytes. For this purpose, we designed and implemented a **shadow memory** that mirrors the actual memory used by the application, thus allowing to store information about memory content.
MemTrace leverages the shadow memory to store state information about every single byte of memory, which can be either *initialized* or *uninitialized*. So, when *MemTrace* detects a read memory access, it can query the shadow memory to check whether the data the program is going to read is initialized or not.
It essentially works as an ideal *hash table*, thus uniquely associating each bit of the shadow

memory to only one byte of the actual process memory, therefore avoiding collisions. Working as an hash table, *update* and *lookup* operations are very fast (i.e., *O(1)*), and the absence of collisions allows to avoid the linear cost due to the presence of multiple elements in the same *bucket*, thus making it very efficient.
Unlike [5], *MemTrace* does not need to cover the whole process address space. Indeed, *MemTrace* is meant to keep track of **uninitialized reads** happening on the stack or on the heap, so it is sufficient to mirror only those memory regions with the shadow memory. Still, there are some difficulties that need to be addressed. First of all, both the stack and the heap may allocate memory pages that are not actually used by the process. This is done simply because the allocation of new memory pages is performed through the invocation of system calls, and is therefore considered an expensive operation. So, in order to avoid allocating memory pages too frequently, a whole block of memory pages are initially allocated for the stack and the heap. If some of these pages are never used by the program, it is not useful to mirror them with the shadow memory. Then, notice that the stack and the heap may require to allocate new pages during program's execution, so, *MemTrace* cannot know on process startup how many pages they will need. Finally, the program may allocate more than a single heap. Therefore, to keep track of the memory accesses performed during execution, *MemTrace* must be able to handle all of them. So, *MemTrace*'s shadow memory is not implemented as a single huge block of sequential shadow addresses, but it is partitioned and allocated on demand. The stack and each allocated heap will be mirrored by their own independent shadow memories, which will be composed of only a few pages at the beginning. If required, then, new pages will be allocated to a shadow memory, and the new page will be made *virtually sequential* with the previously allocated ones by adding them to a vector containing all the shadow pages belonging to the same region. Moreover, the computation of the shadow address only involves some arithmetic operations using division and modulo operators, just like an actual hash computation, thus making it very fast.
Since our objective is try to report *uninitialized reads* that could possibly lead to a leak, not all the uninitialized reads performed during a program's execution are really interesting. Indeed, it may happen that the bytes read by an uninitialized read access are simply loaded into a register, but then they are never used by any instruction, or they are used only by a *cmp* or *test* instruction to evaluate the condition of a branch. In those cases, the uninitialized read cannot lead to a leak.
Moreover, it often happens that the same value is copied in more registers or even in other memory locations. In order to be able to keep track of usages, transfers and copies of uninitialized bytes, *MemTrace* implements a *taint analysis*. Since it requires to keep track of uninitialized bytes, the taint analysis performed by *MemTrace* marks as *tainted* all the bytes that are in the *uninitialized* state when a memory read accesses them, and then it proceeds following the flow of the tainted bytes and marking as *tainted* all of their copies. In summary, the taint analysis has the following main goals:

1. Reduce **false positives** by ignoring uninitialized reads whose bytes are never used by any other instruction (and therefore cannot lead to a leak)
2. Reduce **false negatives** by detecting usages of copies of uninitialized bytes
3. Detect and correctly report **indirect uninitialized reads**

In order to achieve its objectives, the taint analysis required to design a *Shadow Register File*. Indeed, while the shadow memory allows to keep track of loads and stores of uninitialized bytes, it does not allow to follow the propagation of uninitialized bytes within registers. Besides it has an objective similar to the shadow memory, the shadow register file is not implemented in the same way due to some major differences. First of all, registers have a fixed size, so it is not needed to implement the *on demand* allocation of shadow pages, as we can allocate all the space we need on program's startup. Also, the implementation of the shadow register file is actually more complex than the shadow memory. This is because the physical register file of a processor contains many registers which can be very different both in size and in behavior. Moreover, *MemTrace* must also take into account **aliasing** registers. For instance, if a program writes bytes inside register *rax*, also its aliasing reg-

isters will be written, and therefore *MemTrace* must be able to update their state accordingly. The shadow register file hides all the complexities related to different types of registers and aliasing sets, exposing a very simple interface to the other components, which can use it as an intermediary to request to update or query a certain register.
By using both the shadow memory and the shadow register file, the taint analysis is always able to detect and manage usages and copies of uninitialized bytes, thus successfully reducing the number of false positives and false negatives. However, being able of following the flow of uninitialized bytes is still not enough to allow *MemTrace* correctly report the indirect uninitialized reads. Indeed, it is still unable to trace the original memory read access that first loaded the uninitialized bytes from memory. To enable this capability, besides propagating the state of the copied bytes, the taint analysis must also propagate the **origin** of the uninitialized bytes. In the context of taint analysis, we call an *origin* the first memory access that loaded the uninitialized bytes stored in a register or in a memory location.
The most straightforward method to propagate information about origins would be to copy the whole data structure representing the uninitialized read access every time uninitialized bytes are propagated somewhere else. This, however, is not very efficient, as it requires to perform many copies of a complex data structure. For this reason, we implemented a **tag manager**, which is responsible to uniquely associate an integer value, which we called **tag**, to a memory access. The implementation of the tag manager is quite simple, as it mainly consists in a map associating a tag to a memory access. The tag manager also holds a reference count for each tag. The reference count is not really necessary, but it allows to free memory allocated to the association of a certain tag when it is not useful anymore, thus reducing the total amount consumed memory. Given an integer *tag*, the tag manager is of course able to return a reference to the corresponding memory access, so that it is possible, in any moment, to retrieve information about the origin of uninitialized bytes.
Figure 1 shows the basic structure of *MemTrace*.

3. Fuzzing

MemTrace is a dynamic analysis tool and, as such, has an intrinsic limitation: it can only report overlaps detected in the execution paths the program traverses.
In order to partially deal with this limitation, we combined *MemTrace* with *AFL++*[1], which is one of the most effective and widely used fuzzers. By fuzzing the binary, AFL++ will generate a lot of inputs, which can be subsequently used as an input for the program executing it through *MemTrace*.
AFL++ is designed to try and explore as many paths as possible and it will store only those inputs that manage to crash the program and the ones which explore new paths w.r.t. previous executions. So, by using the inputs generated by the fuzzer to execute the program with *MemTrace*, we are able to explore more execution paths. By executing *MemTrace* once for each input generated by the fuzzer, it will generate as many binary reports, that require to be merged in a single human-readable report.
So, we implemented a merging script that extracts information about overlaps from every generated binary report and merges them all in a **merged report**.
Besides helping exploring more execution paths, the fuzzer is helpful also because it allows to detect those uninitialized reads that can be somehow controlled. Indeed, if we want to exploit an uninitialized read to obtain a leak, we would need to control either the content of the memory location it reads from or the memory location itself. So, it would be useless to report uninitialized reads that always read from the same memory location and always read the same bytes.
Since *MemTrace* analyzes executable binaries, it cannot really distinguish which uninitialized reads actually lead to information disclosure. However, by inspecting the merged report and analyzing its overlaps (e.g., through a debugger) the user can see which are the uninitialized reads in the program that can lead to a leak and the reported overlap sets are helpful to understand **if** and **how** it is possible to control them to get **arbitrary** information.
Finally, notice that, as a side effect, *MemTrace* also helps detecting other types of vulnerabilities involving an uninitialized read.

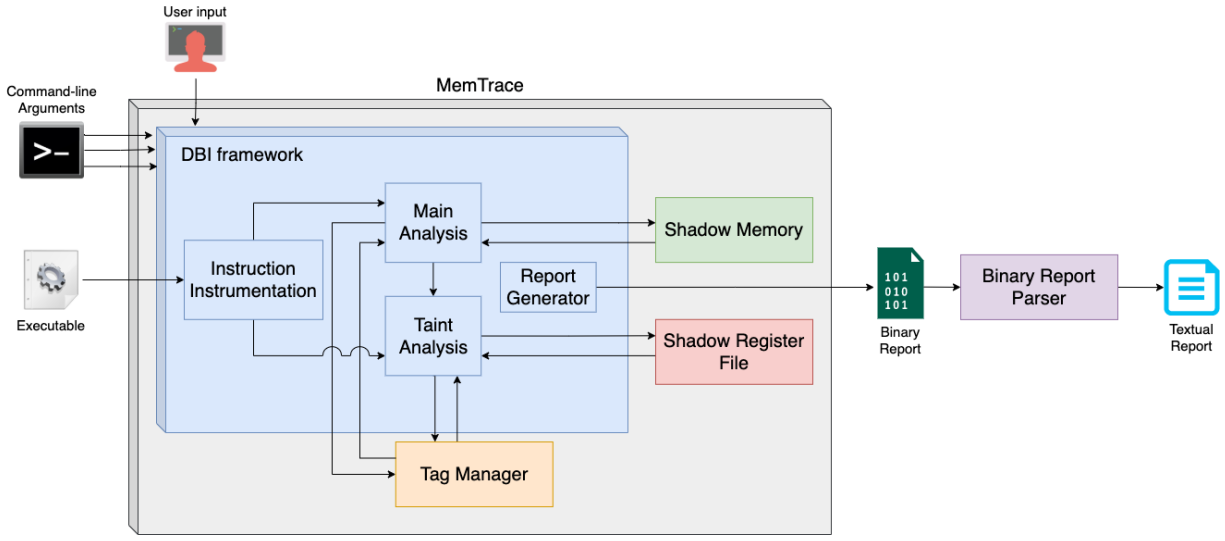

Figure 1: Block diagram of *MemTrace*

Command-line arguments fuzzing

Sometimes the command-line arguments passed to a program may change program's behavior, thus making the execution traverse paths that would not be taken otherwise. In order to deal with this, we leveraged the fuzzer once again.
Extending an example provided within AFL++'s repository, we implemented a library for AFL++ that allows to fuzz also the arguments passed to the fuzzed program. The library simply implements a **hook** for function *__libc_start_main*. So, the call to this function is intercepted and, before calling its original version, some of the bytes generated by the fuzzer are extracted and used as a command-line argument, updating *argc* and *argv* accordingly. Finally, the original *__libc_start_main* is called, passing it the new values of *argc* and *argv*. This way, the fuzzer may help traversing new paths in the executable, thus increasing the coverage.

4. Validation

We performed 2 types of tests: timing and functional.
Since *Memcheck*[6] is a memory error checker also implemented as a DBI tool, we collected the execution times for both *Memcheck* and *MemTrace* and computed the analysis overhead for both the tools to enable a comparison. As a dataset, we used the utilities from package *Coreutils*. From this test, we found out that, despite the additional overhead for each memory access, *MemTrace* is only about 4 times slower than *Memcheck*.
Then, we used *MemTrace* to analyze a set of binaries with known uninitialized read vulnerabilities, and verified whether it was able to detect them and correctly report the memory overlaps. More specifically we tested the tool with:

- 4 *real-world* binaries
- 3 binaries from *Capture The Flag* (CTF) competitions
- 5 binaries from *Cyber Grand Challenge*

When the binary was executed with a manually crafted input that triggered the vulnerability, *MemTrace* was always able to correctly detect and report it. After that, we also tested the effectiveness of the combined execution of *MemTrace* and *AFL++*, and in most of the cases (7 out of 11), this combination successfully detected the vulnerability. Since *MemTrace* always detected the vulnerability with a manually crafted input, we can state that the main limitation of the combined execution is the exploration of the program's CFG. Indeed, we manually investigated the reasons for the failures, and we found out they were mostly due to the strictness of the conditions required to be satisfied to traverse the vulnerable path, thus preventing the fuzzer from generating an input that triggered the vulnerability.
For each executed test, we also manually verified the correctness of all the reported overlaps.

5. Conclusions

The main goal of our work was to design an analysis tool to detect uninitialized reads in a binary executable that might allow to leak information. For this purpose, we developed *MemTrace*, which makes use of dynamic binary instrumentation to keep track of the memory accesses performed by a program and generates a report containing all the memory overlaps, intended as all the uninitialized reads grouped with all the write accesses that overlap the same memory location. In order to let the tool explore paths in a program and discover potential vulnerabilities, we also paired *MemTrace* with *AFL++*.
We tested our tool setting up different types of tests. First, we tested the execution time overhead introduced by *MemTrace* and compared it with the overhead introduced by *Memcheck*, which is implemented in a similar way, showing that *MemTrace*'s overhead is reasonably higher, given the additional amount of operations performed for each memory access.
Then we verified the capability of *MemTrace* to detect uninitialized read vulnerabilities. To do so, we launched *MemTrace*'s analysis on several binaries with a known uninitialized read vulnerability using a crafted input that triggered it and verified that the reports generated by *MemTrace* pointed out the known vulnerability.
Finally, we verified the efficacy of the combined execution with *AFL++* by verifying that the generated report contained the known vulnerability and, in most of the cases, the analysis was able to automatically discover the vulnerability lying in the binaries.
Besides being an immature tool which might be improved in future, the results obtained during testing prove that *MemTrace* is a valuable tool to support the user detecting potential leaks in a binary.

References

[1] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[2] Behrad Garmany, Martin Stoffel, Robert Gawlik, and Thorsten Holz. Static detection of uninitialized stack variables in binary code. 2019.

[3] Christophe Hauser, Jayakrishna Menon, Yan Shoshitaishvili, Ruoyu Wang, Giovanni Vigna, and Christopher Kruegel. Sleak: automating address space layout derandomization. 2019.

[4] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. 40, 2005.

[5] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. June 2007.

[6] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. 2005.