ASanity: On Bug Shadowing by Early ASan Exits

Vincent Ulitzsch
Technische Universität Berlin - SECT
Berlin, Germany
vincent@sect.tu-berlin.de

Deniz Scholz
Technische Universität Berlin - SECT
Berlin, Germany
deniz.scholz@campus.tu-berlin.de

Dominik Maier
Technische Universität Berlin - SECT
Berlin, Germany
dmaier@sect.tu-berlin.de

***Abstract*—Bugs in memory-unsafe languages are a major source of critical vulnerabilities. Large-scale fuzzing campaigns, such as Google's OSS-Fuzz, can help find and fix these bugs. To find bugs faster during fuzzing, as well as to cluster and triage the bugs more easily in an automated setup, the targets are compiled with a set of sanitizers enabled, checking certain conditions at runtime. The most common sanitizer, ASan, reports common bug patterns found during a fuzzing campaign, such as out-of-bounds reads and writes or use-after-free bugs, and aborts the program early. The information also contains the type of bug the sanitizer found. During triage, out-of-bounds reads are often considered less critical than other bugs, namely out-of-bounds writes and use-after-free bugs. However, in this paper we show that these more severe vulnerabilities can remain undetected in ASan, shadowed by an earlier faulty read access.**

To prove this claim empirically, we conduct a large-scale study on 814 out-of-bounds read bugs reported by OSS-Fuzz. By rerunning the same testcases, but disabling ASan's early exits, we show that *almost five percent* of test cases lead to more critical violations later in the execution. Further, we pick the real-world target wasm3, and show how the reported out-of-bounds read covered up an exploitable out-of-bounds write, that got silently patched.

***Index Terms*—memory corruptions, sanitizers, large-scale fuzzing**

I. Introduction

Since the original release of AFL, fuzzing has proven to be extremely effective in identifying memory corruption vulnerabilities. Today, fuzzing is deployed at scale to uncover bugs in software written in memory-unsafe languages. Google's OSS-Fuzz, for example, continuously fuzzes a plethora of famous open-source projects on hundreds of thousands of cores. Since its inception, OSS-Fuzz alone has identified over $40,500$ bugs in 650 open source projects. OSS-Fuzz commonly runs targets with sanitizers, compile time instrumentation tools that facilitate the detection of undefined behavior, e.g., integers overflow, out-of-bounds (OOB) read and OOB write issues.

To unearth memory corruption vulnerabilities, OSS-Fuzz relies on *AddressSanitizer* (ASan). ASan detects OOB memory access in heap, stack, globals, use-after-free bugs, and more [1].

ASan will, by default, abort the code execution immediately after it detects the first violation. However, bugs that result in an out-of-bounds write or a use-after-free access commonly carry higher relevance than bugs that result in an out-of-bounds read. OOB write bugs can overwrite control flow information and potentially result in arbitrary code execution. OOB read bugs' ability to leak information, however, is often considered less critical during triage.

As we will show in this paper, ASan early exits, through aborting execution on the first bug, can *shadow* bugs of higher severity, occurring later in the program's execution.

Consider the following example

```
void swap(char *left, char *right, int len) {
    char tmp[len];
    // Potential OOB read
    memcpy(tmp, left, len);
    // OOB write shadowed by early exit
    memcpy(left, right, len);
    memcpy(right, tmp, len);
}
```

Assume a call to the swap function with the `left` buffer being smaller than the right buffer and the `len` parameter being the size of the `right` buffer. This will trigger two bugs. An out of bounds read in line 6 and an out of bounds write in line 7, with the second being more severe. However, ASan would already cause an exit after detecting the OOB read, shadowing the second bug. This is problematic for anyone assessing the criticality based solely on the ASan report. During triaging, test-cases that seemingly trigger only OOB reads might be dismissed as less interesting, as they would be useless for exploits without additional exploit primitives. In a study conducted by Ding and Le Goues [2], the authors found that around 11% of OOB writes on OSS-Fuzz received a CVE, whereas this number is only around 2% for OOB read issues. As a result, potentially exploitable bugs can be missed. Note that the bug resulting in an OOB write can also be in a completely different part of the program code, and thus would likely remain hidden while triaging the OOB read bug. This underestimated severity might impact the priority assigned to the finding. Thus, severe security vulnerabilities can remain unfixed for an extended period of time and may not be classified as critical vulnerabilities once fixed.

Given the above observations, the natural question to ask is: do ASan early exits impact our bug-finding capabilities in practice, or are they a mere theoretical concern?

Our paper gives an answer in the affirmative – in a large-scale study on OSS-Fuzz, we find that at least 4.5% of OOB read bugs are accompanied by a more severe bug, namely OOB writes or use-after-free accesses. The bugs are triggered

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Digital. Vernetzt." Joint project 6G-RIC, project identification number: 16KISK030.

by the same testcase, but shadowed as a result of ASan early exits.

a) Contributions: This paper answers the following research question: *To what extent do ASan early exists shadow high-severity security bugs?* Our contributions can be summarized as follows.

- We develop ASanity, an open-source framework to scrape and rerun OSS-Fuzz results with arbitrary compiler flags, based on the Monorail scraper by Ding and Le Goues [2].
- Using this framework, we conduct a large-scale study on 814 cases of OOB reads from the OSS-Fuzz database.
- We show that in 4.5% of cases, an input triggering an out-of-bounds read would have actually triggered a more severe bug (use-after-free or OOB write) as well.
- We conduct a case study for one shadowed OOB write vulnerability in the `wasm3` interpreter, proving its exploitability.

Availability

ASanity is built on open-source software. Its source-code and dataset are open-sourced at https://github.com/fgsect/ASanity.

II. Background

A. Fuzzing

Fuzzing is a technique for automatic bug discovery, executing a target application in quick succession with a large number of generated inputs [3]. If one of the inputs, also referred to as a *testcase*, triggers an observable bug, usually a crash or timeout, this testcase is reported and stored for later analysis. Fuzzing has been widely successful in uncovering vulnerabilities and has established itself as standard technique for security researchers. Many coverage-guided fuzzers have emerged in recent years, all mainly targeted at binary applications [3]–[7]. Each gathers feedback about the program's execution, usually about the edges in the control-flow graph an execution took. While implementations differ, the fuzzer generally deems an input interesting if an edge is triggered a different amount of times by the input. The fuzzer stores interesting inputs to its corpus to mutate them further. It then picks a random input from this corpus, applies semi-random mutations to it, and runs the target with the mutated input. If the mutated input triggers new behavior (usually measured in terms of code coverage) the fuzzer again stores the new input to its corpus, continuing the process. Due to its effectiveness, coverage-guided fuzzing is used in favor of other fuzzing methods, wherever it can be used.

B. OSS-Fuzz

OSS-Fuzz, maintained by Google, is a framework for continuous fuzz testing of open source software. As of July 2022, OSS-Fuzz identified over 40,500 bugs in 650 open-source projects [8]. Open-source projects provide a fuzzing configuration, consisting of a Docker build file, which automatically downloads and build the application, and a test-harness, which tests the project's code with inputs generated by the fuzzer. Once a project has successfully been accepted for OSS-Fuzz, it is continuously fuzzed in a distributed execution environment, ClusterFuzz. ClusterFuzz regularly pulls the latest commit of the open-source projects, builds them, and then executes the provided fuzzing-harness. ClusterFuzz is fuzzing on large scale, employing hundred thousand virtual machines to execute the targets under test. ClusterFuzz supports multiple fuzzing engines, namely LibFuzzer, AFL++, and Hongfuzz [9] and complements by compiling and executing the targets with different sanitizers, namely as AddressSanitizer, MemorySanitizer, ThreadSanitizer, LeakSanitizer [10].

The workflow of OSS-Fuzz is as follows. OSS-Fuzz will continuously build fuzz targets, following the instructions given by the developer, and upload these to Clusterfuzz, which will automatically perform fuzzing. OSS-Fuzz will automatically report newly discovered bugs to its own issue tracked, called Monorail. The bug reports include a testcase for the reproduction of the bug and a commit range containing the commit hashes of the project in which the bug has appeared [2]. The reports are first only accessible to the maintainers of the respective project and are then later disclosed to the public, accessible via a web interface.

The bugs reports filed to Monorail also contain the output of the Sanitizer that detected the bug, e.g., the ASan output. From the sanitizer output, ClusterFuzz deduces a recommended security severity estimate. For example, an heap-buffer OOB read is usually assigned a medium severity, while a heap-buffer OOB write is usually assigned a high severity.

C. Address Sanitizer

Applications written in C and C++ are prone to memory corruption bugs. Memory corruption bugs can result in undesired side effects. However, these do not necessarily crash the program. An OOB read from a heap chunk could read values from other chunks or heap metadata, however, it still accesses mapped memory pages. Technically, the memory belongs to the application, but its developer never intended that this line of code could access that part of the memory. Especially during fuzzing, where tens of thousands of executions happen every second, bugs that illegitimately access mapped memory will never be uncovered without additional insight into the program state.

In order to detect these sorts of memory corruption bugs faster, or at all, detect other types of bugs, and to ease the debugging and triaging process, programs can be instrumented using sanitizers. A ubiquitous sanitizer, also used by OSS-Fuzz [9], is *Address Sanitizer*, ASan for short. ASan uses compile-time instrumentation in order to detect memory corruption bugs, for instance out-of-bounds memory access in the heap, stack, globals, and use-after-free bugs [1]. While some works extend on the concept, like QASan, porting it to binary-only targets [11], or works to increase fuzzing performance [12], the goal and underlying implementation is very similar for all of them.

ASan stores fine-grained memory access permissions in shadow memory, and validates all memory accesses at runtime.

By storing the memory access permissions in such a compact way, ASan induces little performance overhead [1]. Each memory byte at address P is mapped to a corresponding shadow memory byte at address

$$(P >> 3) + \texttt{Offset}$$

, where `Offset` is a fixed constant. Each byte in the shadow memory encodes the state of eight bytes in the target's memory. If the shadow byte is set to 0, then all eight bytes are addressable. If the shadow bye is set to a value $k = 1, \ldots, 7$, then the first k bytes of the corresponding application memory are addressable.

During compile time, ASan instruments the target to validate access permissions checks prior to each memory access. ASan also adds *redzones* between each chunk of memory allocated on the heap and each object on the stack. The shadow memory corresponding to these redzones marks as un-addressable, thus, detecting illegitimate out-of-bounds access. To detect use-after-free bugs, ASan *poisons* freed memory chunks, marking them as un-addressable.

Once an illegitimate memory access is detected, ASan aborts the execution of the program under test and reports details about the detected error. Note that ASan can distinguish between different types of illegitimate memory accesses, such as out-of-bounds read, write, or use-after-free bugs.

For example, ASan would print the following report to flag an OOB read error, alongside further information such as stack trace and the state of the shadow map.

```
$ ./helloworld

==29425==ERROR: AddressSanitizer:
heap-buffer-overflow on
address 0x00010520075d
[...]
READ of size 1 [...]
==29425==ABORTING
```

Listing 1: An example of ASan's output, flagging an OOB read issue.

ASan is built into clang as well as gcc, accessible through the `fanisitze=-address` compiler flag.

a) ASan early exits: Notably, ASan aborts the execution of a program after the first discovered illegitimate memory access. As a result, bugs can remain undetected. This behavior can be disabled. Compiling an ASan-instrumented target with the `-fsanitize=recover` compiler flag allows to disable early-exists. Setting the environment variable `ASAN_OPTIONS` to `halt_on_error=false` will prevent the target from crashing upon detecting the first bug. Instead, execution will continue and ASan will report illegitimate memory accesses detected later on as well.

D. Related Work

Ding and Le Goues [2] conduct an empirical study of the bugs identified through OSS-Fuzz. To do so, they first scrape Monorail, OSS-Fuzz's issue tracker, to assemble a dataset of all reported issues on OSS-Fuzz. The paper then analyses the data in multiple dimensions. For example, the work presents the distribution of bug types that are found by OSS-Fuzz, how many bugs are assigned a CVE, and the delay between the introduction of a bug into the codebase and the uncovering of the bug in OSS-Fuzz.

SyzScope can be used to find security critical vulnerabilities from seemingly low-risk fuzzing testcases in the kernel [13]. Given a testcase that triggers a low-risk bug, SyzScope performs further fuzzing campaigns with this testcase, and conducts additional post-processing based on static analysis and symbolic execution to assess whether the testcase can lead to more critical bugs.

Zou, Li, Chen, *et al.* [13] evaluated SyzScope on bugs reported on the continuous fuzzing platform syzbot [14], OSS-Fuzz kernel pendant based on Syzkaller [15]. First, by investigating syzbot's historical data, they found that out-of-bounds write bugs are often fixed much sooner than out-of-bounds read bugs, with 29 days vs 89 days on average attempt-to-fix delay. Using SyzScope, Zou, Li, Chen, *et al.* [13] were then able to show that 183 out of 1,170 low-risk testcases point to bugs with a high security impact.

Fuzzing guides have shown configuration and compiler options to increase ASan's detection capabilities [16]. For instance, Brandt [16] describes that ASan fails to detect semantic OOB accesses in C++ standard library containers because the library automatically grows the container's capacity beyond the allocated size. Using the appropriate compiler flags remediates this issue. Brandt [16] also recommend to use the ASan's recovery option `-fsanitize=recover`, albeit without giving an in-depth explanation or empirical proof.

Jiang, Gan, Herrera, *et al.* [17] present `Evocatio`, a tool to discover *capabilities* yielded by a given crashing testcase/input. The capabilities of a bug comprise information relevant for exploitation, e.g. the bug type (e.g., OOB access, or use-after-free), the access type (read or write access), and the location (e.g., heap or stack). Through a combination of fuzzing and sanitization, `Evocatio` uncovers additional capabilities exposed by a given bug. Specifically, `Evocatio` uses `CapSan`, an extension of ASan, to reveal capabilities exposed by a given testcase. To this end, `CapSan` also makes use of the `halt_on_error=false` option. `Evocatio` discovered additional capabilities for 50% of 38 evaluated real-worlds bugs.

III. Experimental Setup

In order to answer our research question *To what extent do ASan early exists prevent the detection of high-severity security bugs?*, we leverage OSS-Fuzz's findings database. To this end, we develop an open-source analysis framework, *ASanity*. ASanity proceeds in two steps. First, it scrapes the Monorail bug tracker and filters for bugs that have been identified through ASan. We focus on heap-buffer out-of-bounds reads. Second, ASanity recompiles the affected harnesses at the respective commit and executes them with the crashing input provided by OSS-Fuzz. For the recompilation, ASanity additionally includes the `-fsanitize=recover` compiler flag, to disable early-exits later on. For the rerun,

we consequently disable the ASan early-exit by setting the option `halt_on_error=false`. If the crashing input also triggers a heap OOB write or a use-after-free bug, we store the corresponding case for later in-depth analysis.

In order to scrape the Monorail bug tracker, we build on top of the Monorail scraper developed by Ding and Le Goues [2]. The scraper accesses Monorail's web interface through Selenium, a browser automation tool [18]. Monorail's issue descriptions include an excerpt of the ASan output. ASanity parses this output to categorize the issues according to their bug category. Upon rerunning the target, ASanity collects and parses the extended ASan output, now potentially containing more than one bug. Should this output contain a more severe bug than the one described in the respective Monorail issue, ASanity reports the issue to the user. Algorithm 1 summarizes ASanity's workflow.

Algorithm 1 ASanity's workflow

```
I ← Scrape all disclosed issues from Monorail
for each issue i ∈ I do
    Parse ASan report of i
    if i is heap OOB read issue then
        Retrieve and recompile target program P
        Rerun P with testcase of i without early exits
        R ← extended ASan report
        if OOB write or UAF in R then
            Save i as finding
        end if
    end if
end for
```

IV. Evaluation

In order to estimate the prevalence and impact of shadowed bugs, we used ASanity to analyze a large body of OSS-Fuzz issues and conducted a case-study on an underestimated testcase. We present our results in the following.

A. Data Collection

In this section, we present the data we were able to retrieve from OSS-Fuzz. All of the following content has been computed using this data. Our dataset can be summarized as follows.

- We scraped issues reported between May 2016 and May 2022. In total, we successfully received data for 43548 issues.
- The issues spanned 498 different projects in total.
- Out of the retrieved 43548 issues we identified 1986 as heap OOB read issues reported through ASan. Notably, even though only 4.5% of the reported bugs are heap buffer OOB read, these bugs occurred in at least 220 different projects, 44% of all projects that were contained in our dataset.
- Out of 1986 reported OOB read bugs, we successfully reproduced 1788 of them, which equals a 90% success rate. The remaining 180 bugs were not reproducible in our large-scale tests.
- For each of 1788 reproduced OOB bugs, ASanity recompiled the affected project at the respective commit with the compiler flag `-fsanitize=recover` enabled. We achieve a success rate of 55% for these custom builds.
- Finally, our large-scale evaluation successfully ran against 814 different examples of heap OOB read issues, distributed over 159 different projects.

B. Empirical Results

For each of the heap OOB read issues, we ran the recompiled target using the respective crashing input. Table I lists all projects for which at least one testcase triggered an OOB write or use-after-free issue in addition to an OOB read issue. Our key results are as follows.

- For 23 projects (out of 146), we managed to identify at least one testcase that also triggers an OOB write or a use-after-free bug if the `-fsanitize=recover` flag is enabled.
- For 19 projects, at least one testcase that triggered an OOB read also triggered an OOB write.
- For 8 projects, at least one testcase that triggered an OOB read also triggered a use-after-free.
- In total, 38 of the 814 heap OOB read issues also triggered an OOB write or a use-after-free bug. Specifically, 30 of the OOB read issues also triggered an OOB write, and 9 also triggered a use-after-free.
- 65 out of the 814 OOB read issues were flaky, i.e., they only triggered the crash in a non-deterministic manner. None of the flaky crashes triggered an additional OOB write or use-after-free.

In theory, a memory corruption could taint ASan's state, leading to false positives in later error reports, so-called spurious errors [19]. To root out this error source, we validated a sample of 25% of all additional OOB write or use-after-free bugs manually and confirmed their correctness.

C. Case-Study Of A Bug In `wasm3`

a) Bug Analysis: To assess whether the missed out-of-bounds writes really translate into security issues in the real world, we conducted a case study on an OSS-Fuzz bug report for the `wasm3` project [20]. `wasm3` is a WebAssembly interpreter, which takes as input a WebAssembly binary format, parses it, and runs its WebAssembly code. The bug (id 33237 in Monorail) was reported as a heap overflow OOB read in wasm3's `m3_LoadModule` function [21]. Reproducing the testcase with ASan reveals that the out-of-bounds read is triggered by a memcpy operation in the function that parses the data segments of a WebAssembly file, see Listing 2. Note that the parameters `segment->data`, `segment->size`, and `segmentOffset` are user-controlled.

```
1 M3Result InitDataSegments (M3Memory * io_memory,
      IM3Module io_module)
2 {
3     [...]
```

Projects	OOB Reads	OOB Writes	Use-After-Frees
libdwarf	1	1	0
libsass	1	1	0
ghostscript	9	1	0
botan	2	1	0
wasm3	1	1	0
leptonica	16	1	1
mruby	5	1	0
inchi	3	0	1
ffmpeg	58	4	1
openh264	4	2	0
net-snmp	6	1	0
tdengine	3	2	0
muparser	7	3	0
dav1d	2	1	0
libreoffice	20	1	1
libhtp	1	0	1
openjpeg	3	1	0
grok	15	2	0
suricata	4	0	1
ndpi	59	0	2
php	9	2	1
libredwg	18	1	0
libheif	7	3	0

TABLE I: Detailed listing of all OOB read testcases that also triggered other bugs

```
4     i32 segmentOffset;
5     bytes_t start = segment->initExpr;
6     (EvaluateExpression (io_module, &segmentOffset,
       c_m3Type_i32, &start, segment->initExpr +
      segment->initExprSize));
7     if ((size_t)(segmentOffset) + segment->size <=
      io_memory->mallocated->length)
8         u8 * dest = m3MemData (io_memory->mallocated
      ) + segmentOffset;
9         memcpy (dest, segment->data, segment->size);
       //OOB-R here
10    }
11 }
```

Listing 2: The vulnerable ParseDataSegment Function in wasm3.

The testcase provided by OSS-Fuzz triggers an OOB read bug, that reads beyond the boundaries of the `segment->data` buffer. Further analysis of the bug shows that it is triggered by missing bounds checks during parsing of a `section` of the WebAssembly file. The function `ParseSection_Data` does not perform sufficient bounds checks, as seen in Listing 3.

```
1 M3Result  ParseSection_Data  (M3Module * io_module,
      bytes_t i_bytes, cbytes_t i_end)
2 {
3     [...]
4     (ReadLEB_u32 (& segment->size, & i_bytes, i_end)
      );
5     segment->data = i_bytes;
6     i_bytes += segment->size;
7     [...]
8 }
```

Listing 3: The vulnerable function in wasm3. Note that `segment->size` and `segment->data` are user-controlled.

In case the `i_bytes` pointer points beyond the boundary of the data allocated for the file after being increased by `segment->size`, the `memcpy` operation will read beyond the bounds of the file buffer. As a result, ASan will exit, reporting an OOB read bug.

This early-exit, however, hides an OOB write bug that is triggered by the same testcase, also occurring during the memcpy in the `InitDataSegments` function. The root cause of this OOB write is that the `InitDataSegments` function suffers from an integer overflow vulnerability. The `segmentOffset` variable is parsed as a signed integer and then cast into a `size_t` when performing the bounds check in line 7 of Listing 2. If the attacker provides a negative `segmentOffset`, then this type cast will overflow – providing a segment of size at least $|segmentOffset|$ will cause the addition in line 7 to overflow again, resulting in the bounds checks being passed. If the segmentOffset provided was negative, however, line 8 will result in a write *before* the dest buffer. This gives the attacker a primitive to overwrite data and metadata of other chunks on the heap. As we will show below, this OOB write is a severe security vulnerability that leads to arbitrary code execution.

We highlight that the OOB read and the OOB write issue are caused by two different bugs. The OOB read is caused by an insufficient bounds check, while the OOB write is caused by an integer overflow. The testcase provided with the bug report for `wasm3` triggers both bugs by coincidence — they are not correlated. To fix the OOB read bug, the developers introduced a bounds check and error out in case of a violation, see Listing 4.

```
M3Result  ParseSection_Data  (M3Module * io_module,
     bytes_t i_bytes, cbytes_t i_end)
{
     [...]
     (ReadLEB_u32 (& segment->size, & i_bytes, i_end)
     );
     segment->data = i_bytes;
     i_bytes += segment->size;
     _throwif("data segment underflow", i_bytes >
     i_end);
     [...]
}
```

Listing 4: The fixed `ParseSection_Data` function with an included underflow check.

After applying only this fix, the `wasm3` will then error out when parsing the crashing testcase, never executing the memcpy statement that triggered the OOB write issue. A regression test with the crashing testcase would thus not detect the OOB write issue, although it is still present in the code. Thus, if a developer would follow a typical bugfix workflow of first seeing the ASan early-exit report for issue 33237, fixing the issue by adding the bounds check, and then running regression tests, the OOB write bug would still remain in the codebase. However, by disabling ASan early-exits, the developer would notice both the OOB read as well as the OOB write report, again illustrating the importance of leveraging this option during triage. The `wasm3` developers indeed fixed the OOB write and OOB read bug in two different commits, however,

both were labeled with "*Fix OSS-Fuzz issues*". Reconstructing their exact workflow is out of the scope of this work.

b) Exploitability: To illustrate the security of shadowed bugs, we present an exploit for the OOB write triggered by the testcase provided with OSS-Fuzz issue 33237. To exploit the OOB write vulnerability present in the older version of the `wasm3` interpreter, we prepare a malicious WebAssembly binary file. Parsing the file with the vulnerable `wasm3` interpreter will lead to code execution and spawn a shell. We construct the file as follows.

1) We prepare the heap by declaring multiple WebAssembly functions – each function declaration results in one independent allocation on the heap.
2) In doing so, we groom the heap so that the heap object corresponding to the function `_start`, which is called by default, is placed exactly before the chunk allocated for the `dest` buffer. Note that the function object contains a function pointer `function->compiled`, which we can overwrite with arbitrary data abusing the out-of-bounds of write.
3) This function pointer is then deferenced in the `RunCode` function, calling the function that `function->compiled` points to.

```
#define nextOpImpl() ((IM3Operation)(* _pc))(
    _pc + 1, d_m3OpArgs)
[...]
d_m3RetSig  RunCode  (d_m3OpSig)
{
    nextOpDirect();
}
```

Listing 5: The RunCode Function of Wasm3.

4) We set `function->compiled` pointer to point to our payload on the heap, which, in turn, contains a pointer to an `execve` gadget in the libc – executing this gadget will spawn the desired shell.

Listing 6 contains an excerpt of the Python script used to assemble the exploit. The Python script assembles WebAssembly-Text file, which we then convert to a WebAssembly binary payload using the `wat2wasm` tool from the WebAssembly Binary Toolkit [22].

```
import subprocess
from pwn import *
payload_len = 160
target_gadget = p64(0x7ffff7ea5b01)
our_address = 0x5555555f0018# 0x5555555f2428
offset = -160 # function pointer at dest-160
payload = flat({0: p64(our_address+8), 8:
    target_gadget, 56: "_start\x00"},length=
    payload_len, filler='\x00')
wat_text = '''
(module $M1
'''
for i in range(86):
    wat_text += f'''
    (func $_fillfunction{i} (param i32) (param i32)
    )
    (export "fillfunction{i}" (func $_fillfunction{i
    }))
    '''

wat_text += '''
(func $_start (param i32) (param i32) )
(export "_start" (func $_start))

'''
wat_text += b'''
(data (i32.const %d) "%b")
)''' % (offset,payload)
[...]
```

Listing 6: The Python script that assembles the payload of our exploit

Invoking the vulnerable `wasm3` interpreter with the crafted file will trigger the exploit and spawn a shell.

V. Future Work

A. Relaxed ASan Fuzzer

Current fuzzing campaigns have two choices: use ASan, and exit early, or run without address sanitization, and potentially not find these memory corruptions at all. Fuzzers running on ASan instrumented targets use ASan in its early-exit mode, resulting in missed bugs of higher criticality, as we show throughout this paper. A relaxed ASan Fuzzers could instead disable ASan's early-exit behavior, leading to the detection of additional bugs and increased code coverage. Without ASan early-exit behavior, testcases that cause memory violations do not necessarily crash the target. This can be compensated for by a) either additional seed queue post-processing that includes parsing the ASan output, or b) by modifying ASan to *defer* a crash until the program exits, instead of either crashing immediately or not crashing at all. The additional post-processing could even uncover multiple bugs per testcases, including shadowed ones. We note that disabling early-exit behavior might also result in further mutation of crashing testcases, which are usually not considered for further mutation. Doing so trades fuzzer attention from non-crashing testcases for further mutating crashing testcases. This trade-off is favorable in situations where different bugs are triggered by two similar inputs, or to uncover additional bug capabilities, c.f. [17]. Repeatedly executing crashing testcases also incurs a performance penalty, as the ASan reporting entails additional overhead. This performance penalty will however be negligible in most cases, as the crashing testcases will usually only make up a tiny fraction of processed inputs.

B. Mutate Testcases for Shadow Bugs

Right now, we only re-run the crashing testcases from OSS-Fuzz. However, we note that these testcases may not immediately lead to shadowed bugs. Jiang, Gan, Herrera, *et al.* [17] show that some testcase only trigger an OOB read, but are only *a few mutations away* from triggering a more severe vulnerability as well. ASanity could be extended to quantify how many testcases that trigger OOB read issues result in more severe vulnerabilities when mutating the testcases further. To this end, a future study could run hyper-focused fuzzing campaigns on each target, using a relaxed ASan fuzzer that is seeded with only the OOB read testcases. Alternatively, ASanity could make use of existing tools for bug evaluation,

such as `Evocatio` [17]. We assume this would identify an even higher number of shadowed bugs.

C. Extend ASanity Analyses

During the course of this paper, we specifically targeted heap corruptions. However, this leaves room for further analyses as valuable future work. Specifically, ASanity's analyses can be extended to other bug classes, such as stack-based buffer overflows, or even other sanitizers.

VI. Conclusion

Our large-scale study shows that ASan early-exists can lead to an underestimation of a finding's criticality. Critical bugs may be shadowed by earlier, less severe OOB read bugs that may or may not be related. We specifically aimed to find such cases, where testcases that trigger exploitable bugs were classified as heap OOB read issues and managed to exploit such a case successfully. The exploitable bug was not tagged as a critical security vulnerability.

As actionable advice,we recommend running ASan with the `-fsanitize=recover` option during the triage phase. This will increase the chance of detecting severe security bugs without impeding fuzzing performance or negatively impacting the triage phase. In light of the results of the present study, we expect that integrating these changes in (large-scale) fuzzing setups will help uncover severe security vulnerabilities that would otherwise remain hidden.

Acknowledgment

The authors wish to thank Fabian Freyer for valuable input.

References

[1] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA: USENIX Association, Jun. 2012, pp. 309–318, ISBN: 978-931971-93-5. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[2] Z. Y. Ding and C. Le Goues, "An empirical study of oss-fuzz bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 131–142. DOI: 10.1109/MSR52588.2021.00026.

[3] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, 2020, pp. 10–10.

[4] R. Swiecki, *Honggfuzz*, Nov. 2018. [Online]. Available: https://github.com/google/honggfuzz.

[5] M. Zalewski, *American Fuzzy Lop - Whitepaper*, https://lcamtuf.coredump.cx/afl/technical_details.txt, [Online; accessed 10 April. 2022], 2016.

[6] LLVM Project, *libFuzzer – a library for coverage-guided fuzz testing.* https://llvm.org/docs/LibFuzzer.html, [Online; accessed 10 April. 2022], Sep. 2018.

[7] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1051–1065.

[8] Google, *Oss-fuzz: Continuous fuzzing for open source software*, GitHub Repository, Jul. 5, 2022. [Online]. Available: https://github.com/google/oss-fuzz (visited on 08/07/2022).

[9] Google. "Oss-fuzz." (Jan. 19, 2022), [Online]. Available: https://google.github.io/oss-fuzz/ (visited on 08/08/2022).

[10] Google, *Sanitizers*, GitHub Repository, Nov. 3, 2020. [Online]. Available: https://github.com/google/sanitizers (visited on 08/08/2022).

[11] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *2020 IEEE Secure Development Conference (SecDev)*, 2020.

[12] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 4345–4363, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen.

[13] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "{Syzscope}: Revealing {high-risk} security impacts of {fuzzer-exposed} bugs in linux kernel," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3201–3217.

[14] Google, *Syzbot*, 2023. [Online]. Available: https://syzkaller.appspot.com/upstream/.

[15] Google, *Syzkaller*, 2023. [Online]. Available: https://github.com/google/syzkaller.

[16] L. Brandt, *One weird trick to improve bug finding with asan*, https://landaire.net/one-weird-asan-trick/, [Online; accessed 11. February. 2023], 2023.

[17] Z. Jiang, S. Gan, A. Herrera, *et al.*, "Evocatio: Conjuring bug capabilities from a single poc," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1599–1613.

[18] SeleniumHQ. "Webdriver." (2021), [Online]. Available: https://www.selenium.dev/documentation/webdriver/ (visited on 08/08/2022).

[19] Google, *Addresssanitizerflags*, GitHub Repository, May 15, 2019. [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizerFlags (visited on 2022).

[20] V. Shymanskyy, *Wasm3*, 2023. [Online]. Available: https://github.com/wasm3/wasm3.

[21] *Oss-fuzz issue report 33237: Heap-buffer-overflow in m3_loadmodule*, 2021. [Online]. Available: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=33237.

[22] WebAssembly, *The webassembly binary toolkit*, 2023. [Online]. Available: https://github.com/WebAssembly/wabt.