Unlimited null: achieving memory safety by extending memory protection

hikari_no_yume

2023-04-00

We present a novel approach to preventing memory safety vulnerabilities that extends existing memory protection systems. Our method achieves complete memory safety, while improving on existing approaches by being simple to implement, not requiring changes to application software or hardware, and having a positive impact on code size and performance.

1 Background and motivation

Memory safety has become an increasingly pressing security and reliability issue in computer software. While approaches like Rust[1] and CHERI[2] offer promising solutions for future software and hardware, they can't easily be applied to existing systems. Consequently, while such approaches are successfully reducing the number and severity of vulnerabilities in new systems,[3][4] there is a long tail of existing systems where memory safety issues continue to be discovered and exploited, and they continue to cause a large proportion of security vulnerabilites.[5]

2 Virtual memory and memory protection

Virtual memory is a ubiquitous feature of modern computer systems.[6] One of its many benefits is that regions of virtual memory can be *protected* from unintended accesses (that is, reads and writes), so-called memory protection. This protection is independent of the programming language, so even notoriously unsafe languages like C can benefit. However, prior work has not realised the full potential of this capability. Indeed, even though almost all modern systems make use of memory protection, memory safety vulnerabilities continue to exist.

There are many common techniques that use memory protection to prevent or mitigate certain kinds of memory safety issues. One technique that is of particular interest to us is the so-called *null page* or *page zero*. This is a region of the virtual address space that starts at `0x0` (i.e. null), is at least a single page in size, and which has memory protection applied so that all accesses are disallowed. The purpose of this region is to trap null-pointer accesses: if memory-unsafe code attempts to dereference a null-pointer, it will usually resolve to an address within this region, and therefore can be trapped by memory protection. This is a standard feature of modern operating systems and has shown itself to be effective at catching many, but not all, null pointer accesses.

3 Limitations of conventional null page design

The principal limitation of the conventional null page is its size. An access to a pure null pointer will always be trapped, but often there is some address calculation involved that leads to the access landing outside the protected region. In pseudo-C: `*(int*)NULL` will always hit the null page, because the resulting address is always `0x0`, but whether `((Foo*)NULL)->bar` hits the null page depends on the offset of `bar` within the struct `Foo`. On 32-bit iOS, for example, the null page size is 4KiB by default, so if `bar` were at an offset of 4KiB or greater, the access wouldn't be trapped.

While recent work has significantly pushed this limit (for example, Darwin increases the null page size to 4GiB on 64-bit targets[7]), it is not completely removed, so the null page has remained a mere mitigation, rather than a complete protection against null-pointer issues.

4 Our contribution

In other words, the limitation of the conventional technique is:

> If only part of memory is protected against null-pointer accesses, such accesses will still be possible in some circumstances.

By divine revelation, we obtained the following corollary:

> If all of memory is protected against null-pointer accesses, no accesses will be possible in any circumstances.

We consider that this is a novel insight that can be practically applied. In order to work backwards to our pre-ordained conclusion, we ask a rhetorical question: if we wish to prevent all unsafe use of memory, why is only some of memory protected?

[1]The Rust Core Team. (2015). Announcing Rust 1.0. *blog.rust-lang.org.*

[2]Watson, R. N., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., ... & Vadera, M. (2015, May). CHERI: A hybrid capability-system architecture for scalable software compartmentalization. *2015 IEEE Symposium on Security and Privacy* (pp. 20-37). IEEE.

[3]Xu, H., Chen, Z., Sun, M., Zhou, Y., & Lyu, M. R. (2021). Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *31*(1) (pp. 1-25).

[4]Stoep, J. V. (2022, December). Memory Safe Languages in Android 13. *security.googleblog.com.*

[5]Miller, M. (2019, February). Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. BlueHat IL.

[6]Denning, P. J. (1997). Before memory was virtual.

[7]Apple Inc. (2020, September). ld(1). *Darwin General Commands Manual.*

With this in mind, we propose a new technique: the *unlimited null page*. By simply extending the "null page" region to cover the entire virtual address space, all null-pointer accesses can be trapped. This can be trivially implemented in existing operating system kernels without requiring hardware or application software changes.

As we will see in later sections, the potential for this technique extends far beyond preventing null-pointer accesses.

5 Prototype

In order to test the viability of this technique, we have implemented it in a real-world system. We chose to implement it in a fork of *touchHLE*,[8] a high-level emulator for iPhone OS applications. touchHLE was chosen because it already implements null-page memory protection, is easy to modify, and *only* runs real-world existing applications;[9] the fact that it is also the author's personal hobby project was surely not a factor.

While touchHLE is written in Rust, it is only the emulator itself which has been memory-safe until now; the protection available to emulated apps has been constrained by the limits of conventional techniques, and in fact was inferior to the original platform, iPhone OS. By implementing our new approach, we hoped to not only reach comparable safety, but surpass it.

We found that our approach was very practical to implement: only 4 files had to be changed, 13 lines inserted and 28 lines deleted, with almost no refactoring required.

In order to assess the functional impact of our changes, we created a corpus from 287 randomly selected iPhone OS applications. Three applications were removed due to [note to self: insert post-hoc justification]. We then ran the unmodified and modified versions of the emulator against this selection of 284 applications.

We found that 284 of 284 applications immediately crashed in our modified version of the emulator. On the other hand, with the unmodified emulator, 284 of 284 applications crashed. This is a difference of 0, so we conclude that there is no impact on application compatibility.

We also observed significant performance and code size improvements. Executing the entire corpus in the modified emulator took 1m33.524s in total wall-clock time, down from 3m38.580s in the original, a reduction of circa 57%. The size of the optimised release binary for the emulator shrank from 12.37MB to 11.44MB, a reduction of circa 7.5%. We hypothesise that these improvements come from the Rust compiler removing code that contained security vulnerabilities, now statically provable to be dead.

[8]https://github.com/hikari-no-yume/touchHLE/tree/unlimited-null

[9]hikari_no_yume. (2023, March). Apps supported by touchHLE. *touchHLE project.*

6 Other memory safety issues

While our original goal was to prevent null-pointer accesses, we have discovered that the unlimited null page approach can, in fact, prevent a diverse range of other memory safety issues. In a particular order:

- All issues that *write XOR execute* permissions attempt to mitigate.
- All issues that *guard pages* attempt to mitigate.
- Stack and buffer overflows.
- Integer overflow during address calculation.
- Stack and heap corruption.
- Uninitialised memory.
- Use after free.
- Double free.
- Heap fragmentation.

To our amazement, it can even prevent safety issues in broader categories:

- Time of check to time of use vulnerabilities.
- Logic errors.
- Denial of service vulnerabilities.
- Backdoors, insider attacks and other malicious activity.

In fact, no safety issue the authors are aware of can be reproduced in a system using unlimited null pages, nor any safety issue we aren't aware of. It trivially follows that this system achieves complete memory safety.

7 Comparisons

Environment	Protected range	Protected fraction
MS-DOS	None	0
iPhone OS (32-bit)	`[0, 0x1000)`	9.54×10-7
macOS (64-bit)	`[0, 0x100000000)`	2.33×10-10
touchHLE (*ours*, 32-bit)	`[0, 0x100000000)`	1

Solution	Existing code	Existing hardware	Memory safety
None	Yes	Yes	No
CHERI	Yes	No	Partial
Rust	No	Yes	Outside `unsafe`
Ours	Yes	Yes	Yes

8 Future work

While our approach achieves complete memory safety by trapping all invalid memory accesses, it has been noted that in some situations, some or all *valid* accesses could also be trapped, which theoretically has performance or correctness implications. We welcome future work that can avoid this issue.

A more significant limitation is the unknown applicability of our work to computing systems that do not use random access memory. We hope to soon publish a follow-up paper on this topic, tentatively titled *Mission Impossible: achieving memory safety on Turing machines with self-destructing tape.*