Poster: WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches [1]

Jianhao Xu*†, Luca Di Bartolomeo†, Flavio Toffalini†, Bing Mao*, Mathias Payer†
jianhao_xu@smail.nju.edu.cn, {luca.dibartolomeo, flavio.toffalini}@epfl.ch, maobing@nju.edu.cn, mathias.payer@nebelwelt.net
State Key Laboratory for Novel Software Technology, Nanjing University, EPFL†*

***Abstract*—Code-reuse attacks are dangerous threats that attracted the attention of the security community for years. These attacks aim at corrupting important control-flow transfers for taking control of a process without injecting code. Nowadays, the combinations of multiple mitigations (e.g., ASLR, DEP, and CFI) drastically reduced this attack surface, making running code-reuse exploits more challenging.**

Unfortunately, security mitigations are combined with compiler optimizations, that do not distinguish between security-related and application code. Blindly deploying code optimizations over code-reuse mitigations may undermine their security guarantees. For instance, compilers may introduce double-fetch vulnerabilities that lead to concurrency issues such as Time-Of-Check to Time-Of-Use (TOCTTOU) attacks.

In this work, we propose a new attack vector, called WarpAttack, that exploits compiler-introduced double-fetch optimizations to mount TOCTTOU attacks and bypass code-reuse mitigations. We study the mechanism underlying this attack and present a practical proof-of-concept exploit against the last version of Firefox. Additionally, we propose a lightweight analysis to locate vulnerable double-fetch code (with 3% false positives) and conduct research over six popular applications, five operating systems, and four architectures (32 and 64 bits) to study the diffusion of this threat. Moreover, we study the implication of our attack against six CFI implementations. Finally, we investigate possible research lines for addressing this threat and propose practical solutions to be deployed in existing projects.

1. Background

Most software running on today's systems is written in low-level languages like C or C++. As these languages are prone to memory corruption bugs that often enable attackers to launch powerful code execution attacks, the applications need to be thoroughly tested to remove as many bugs as possible. As testing is generally incomplete due to state explosion, mitigations are added to the code to make exploitation more challenging [2].

Widely deployed mitigations such as Address Space Layout Randomization (ASLR), stack canaries, and data execution prevention (DEP) lower the exploitability of bugs. However, code-reuse attacks such as Return-Oriented Programming (ROP) remain effective under all these mitigations. These attacks redirect control flow to execute legitimate instruction sequences in program memory for malicious purposes.

Control-Flow Integrity (CFI) [3], widely recognized as a key mitigation to stop code-reuse attacks, restricts control-flow transfers to strictly follow some benign program execution. Specifically, CFI first statically computes the program's control-flow graph (CFG) and determines all legitimate targets of control-flow transfers. Then, CFI instruments the code with checks to validate each control-flow transfer at run-time. CFI is practical: CFI implementations are readily available in production compilers and have been widely deployed, with some trade-offs between granularity and performance. For example, the Android ecosystem deployed LLVM-CFI [4] in Android 9's kernel [5], and Windows applications are widely protected by Microsoft's Control Flow Guard [6].

Despite the advance of modern CFIs, they cannot stop all the attacks. In practice, not all code may be protected through CFI [7] since their precision is limited by the target sets size. Still, the key advantage of CFI is that the size of the target set is generally small (between many targets with under ten targets and few targets at around 100 targets even for complex applications) [8]. CFI is therefore believed to practically stop code reuse attacks or, at least, make it almost impossible to bypass.

Mitigations are generally implemented as compiler passes (or, rarely, as static binary rewriting passes). However, compilers currently are *unaware of security checks* inserted by mitigations and consider them just code that undergoes the same optimizations as all non-privileged code.

2. WarpAttack [1]

In this work, we present WarpAttack: a novel attack that bypasses strong anti-code-reuse mitigations and grants arbitrary code execution to adversaries. Our attack exploits a misalignment between CFI implementations and assumptions of C/C++ compilers when translating switch statements to jump tables (or multiple checks against one target). The CFI threat model assumes that adversaries can modify arbitrary data including function pointers but not code. Therefore, CFI implementations assume that a combination

of storing jump tables in read-only memory and bound checking the indirect control flow transfer will sufficiently protect the jump tables. However, compilers are unaware of CFI's security assumptions and optimize code, which results in double fetch vulnerabilities and TOCTTOU attacks between the bound check and the indirect jump. The code for computing indirect jumps in switch statements often follows this pattern: given a jump table and an index, the program first fetches the index value to validate the bound check against the jump table size, then, it fetches the index value again for processing the actual jump. Since the index is loaded (fetched) multiple times, an adversary may race against the bound check and overwrite the index value after the bound check itself, finally allowing arbitrary jumps outside the jump table. Similar issues were investigated by previous reports [9], but, so far, were considered a concurrency bug.

Figure 1 illustrates the timeline of the attack. We use x86/64 assembly-like pseudo code to represent the victim code. Note that real victim code may be more complex but should include all the components shown here.

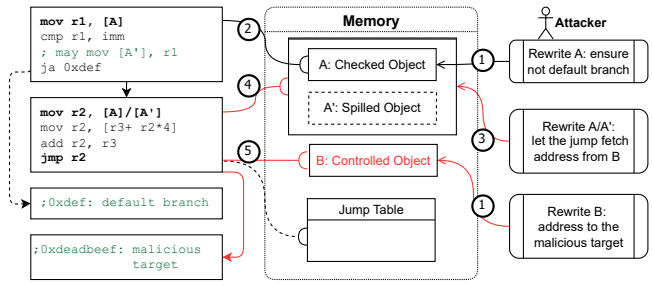

Figure 1. The timeline of exploiting the double-fetch to hijack control flow. `r1,r2,r3` means registers, `imm` means some random value and `[A]/[A']` means memory access of `A` or `A'`.

3. Evaluation

To evaluate our new attack WarpAttack, we implement (and release as open-source) a proof-of-concept attack against Firefox to demonstrate WarpAttack is practical under a realistic adversary model. Our PoC shows how to win the race condition and take control of the victim program. Additionally, we implement a lightweight binary analysis to detect compiler-introduced double-fetches in widely used C/C++ programs on multiple platforms (with a false-positive rate of 3%). Our results show that (i) mainstream compilers including Clang, MSVC, and GCC generate such victim code patterns in sensitive situations, (ii) most CFI implementations are vulnerable to our attack, (iii) widely used C/C++ programs like Firefox and Chrome present such victim code patterns; (iv) these code patterns can be found on different platforms including Linux, Windows, and Mac OS and cross-architecture (i.e., x86/64, ARM, MIPS, RISCV). Finally, we discuss possible mitigations and propose future research directions to cope with WarpAttack.

In summary, our contributions are:

- We introduce WarpAttack, a new attack to bypass code-reuse mitigations and grant arbitrary code execution to adversaries.
- We show a working proof-of-concept example of WarpAttack against Firefox.
- We conduct a comprehensive research on compiler-introduced double-fetch code samples in popular applications across multiple OSs and architectures.
- We perform a study of different compilers and CFI implementations to demonstrate the magnitude of our attack has real-world impact.
- We design (and implement) a new lightweight binary analysis to detect victim code (that we release as open-source).

4. Mitigation Options

The most direct way is to prevent compilers from producing gadget code via options like `-fno-switch-tables` (GCC). Another promising mitigation is to add dynamic checks for every indirect jump including the `switch` ones.

References

[1] J. Xu, L. D. Bartolomeo, F. Toffalini, B. Mao, and M. Payer, "Warpattack: Bypassing cfi through compiler-introduced double-fetches," in *44th IEEE International Symposium on Security and Privacy (IEEE S&P 23)*, 2023.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, 2005, pp. 340–353.

[4] L. team, "Clang 16.0.0git documentation: Control flow integrity," *https://clang.llvm.org/docs/ControlFlowIntegrity.html*, 2022.

[5] A. Docs, "Kernel control flow integrity," *https://source.android.com/devices/tech/debug/kcfi*, 2022.

[6] M. Docs, "Control flow guard for platform security," 2022, https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[7] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks," in *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, vol. 22, 2022, pp. 1–17.

[8] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[9] F. Wilhelm, "Xen xsa 155: Double fetches in paravirtualized devices," 2020, https://insinuator.net/2015/12/xen-xsa-155-double-fetches-in-paravirtualized-devices/.