1. Introduction

Most memory corruption exploits are aimed at disrupting the control-flow of a program. State-of-the-art Control-Flow Integrity (CFI) [1] preservation proposals, such as Intel’s Control-Flow Enforcement Technology (CET) [2] make it significantly harder for an attacker to gain arbitrary code execution, drastically shrinking the existing attack surface. With such defenses in place, an attacker cannot directly tamper with saved function return addresses: they must target other control variables. Function pointers constitute a possible exploitation target, as overwriting one allows to redirect the control-flow of a program without triggering CFI defenses.

We focus on global function pointers defined in C libraries whose calls are reachable from exported functions, which implies they can possibly also be reached from the code of programs using the library. A memory corruption vulnerability that provides an arbitrary write primitive could be leveraged in such programs to gain arbitrary code execution through the overwrite of the right global library function pointer. This kind of attack is complicated by the need of identifying global function pointers and the constraints to be satisfied in order to trigger a call to them from library code, such as values of function parameters and other global variables, which involves complex and time-consuming manual inspection of library source code.

We present an approach based on static analysis and symbolic execution to automate this process. We developed UNTANGLE\(^1\), an open-source tool implementing this approach with the goal of aiding global function pointer hijacking in C libraries, using CodeQL, the angr symbolic execution engine [3, 4], and an ad-hoc symbolic memory model specialized in dealing with complex structure pointer parameters. Analyzing several open source C libraries, we identify 57 different global function pointers whose calls are reachable from a total of 1488 unique exported library functions, verifying the validity of the found solution for 484 of them.

2. Background

2.1. Exploitation Techniques and Defenses

Memory corruption exploits and defenses against attacks targeted at exploiting mem-
ory corruption vulnerabilities have significantly evolved in the last two decades. A non-negligible portion of exploits targets the stack of a program to overwrite the currently executing function’s saved return address through stack buffer overflows [5, 6].

Redirecting the control-flow to shellcode previously injected into the target program as a means of gaining arbitrary code execution is a possible way of exploiting such a vulnerability. The introduction of stack canaries and Write Xor Execute (W⊕X) memory protection policies mitigates these kind of attacks, and as a result, code-reuse techniques such as Return-Oriented Programming (ROP), Jump-Oriented Programming (JOP) and Call-Oriented Programming (COP) took hold [7–9]. The introduction of Address Space Layout Randomization (ASLR) makes code-reuse attacks more difficult, but still possible under the assumption of being able to extract information about the layout of the victim program’s address space.

More sophisticated software and hardware solutions for CFI preservation were developed, among which is Intel’s CET, one of the most recent and advanced. CET provides CPU instruction set architecture extension allowing software to easily set up hardware defenses against control-flow hijacking. It introduces two main features: the use of a shadow stack to provide saved return address protection, defending against ROP, and Indirect Branch Tracking (IBT) to prevent misuse of indirect branch instructions, typical of JOP, COP and similar techniques.

With advanced CFI enforcement measures such as CET in place, redirecting the control-flow of a program targeting the saved return address is unfeasible, and exploitation can only take place carefully targeting other kinds of control data, such as function pointers.

2.2. Function Pointer Hijacking

Common reasons for function pointer usage in library code are providing the user with runtime hooks for particular function invocations, implementing function callbacks, and delivering notifications for asynchronous runtime events. Even in the presence of strong defense mechanisms such as CET, if the right conditions are met, overwriting a function pointer used by the program with the address of an arbitrary function can be done without failing CFI checks. The ability to control the parameters supplied to the pointed function could however be necessary and depends on the specific case at hand. We focused on the hijacking of global function pointers defined in C libraries as a possible exploitation entry point for a program using them.

2.3. Static Source Code Analysis

Several existing automated platforms and tools leverage static source code analysis to detect errors, common programming mistakes, anti-patterns, bugs and security vulnerabilities [10].

CodeQL is a powerful static analysis framework supporting multiple languages, including C, which works on an ad-hoc database built for a given code base, giving the ability to perform queries in a language similar to SQL. CodeQL is used by GitHub to perform automatic auditing and vulnerability detection for code hosted on their website. In our work, CodeQL is used to identify global function pointer variables and analyze the call graph of a C library to check whether any calls to such function pointers are potentially reachable from exported library functions.

2.4. Symbolic Execution

Symbolic execution is a *dynamic* program analysis technique in which the program to be analyzed is driven through its execution by a specialized engine, with the goal of understanding the inputs needed for the program to produce a specific output or execute a specific part of its code. The symbolic engine feeds the program with *symbolic inputs* (sequences of boolean variables called *bitvectors*), rather than obtaining concrete inputs, and is able to manage multiple program states, keeping track of the symbolic expressions accumulated along the way for each state. Whenever the program needs to branch based on the value of symbolic data, the current state is duplicated, and execution continues from both sides of the branch, taking into account the constraints necessary for each side. These constraints can then be tested for satisfaction and evaluated to determine concrete values of the symbolic inputs that given to the program would lead execution to reach a given state.

A critical aspect of the design of a symbolic
The execution engine is its memory model, which dictates how to emulate load/store memory operations of analyzed programs. The symbolic execution engine we used is angr, which employs a fully symbolic memory model, but also permits to extend or override its logic with a dynamic plugin system.

3. State of the Art

The introduction of CFI enforcement measures like CET stimulated the development of attacks targeting non-control data: sophisticated data-oriented attack techniques, along with tools that help automate exploitation, have been proposed in recent years. Data-Oriented Programming (DOP) [11] is a technique to construct expressive non-control data exploits. It allows an attacker to perform arbitrary computations in program memory by chaining the execution of short sequences of instructions, called DOP gadgets. The downside is that the gadget chains must be crafted by hand.

Block-Oriented Programming (BOP) [12] is a further improvement of data-oriented attacks: it uses basic blocks as gadgets and leverages symbolic execution to automatically find the constraints on variables and memory-resident data needed to redirect the control-flow. BOP attacks are specifically aimed at creating a chain of basic blocks that does not trigger CFI preservation mechanisms, and since they do not overwrite the saved return address, they can bypass shadow stacks too. The advantage of BOP, with respect to DOP, is that the gadget chain building process is automated.

To the extent of our knowledge, no existing work explores the automation of both global function pointer identification and hijacking in C libraries. Most of the existing work and research focuses on subsequent exploitation steps instead. In particular, the tool presented by [12] to automate the creation of BOP-chains requires an entry point to start the analysis from, for which a hijacked function pointer would make a good candidate.

4. Problem statement

4.1. Threat Model

Our exploitation scenario considers a program that is running on a machine employing state-of-the-art control-flow hijacking defenses, such as fully enabled Intel CET. Moreover, the program is also protected through stack canaries, W⊕X memory protection policies and ASLR. We assume that the program uses functions exported by a C library (statically linked or dynamically loaded at runtime) that contain, or can lead to, calls to global function pointers defined within the library itself.

We assume that the program presents a known memory corruption vulnerability that can lead to an arbitrary memory write. We also assume that the attacker is able to discover, for example thanks to an information leak, the base address at which the target library was loaded under ASLR. We believe these assumptions to be realistic and practical, in line with the ones of the same mechanisms that aim at preventing control-flow hijacking.

4.2. Problem Definition

We focus on the problem of finding calls to global function pointers in the source code of a given C library and identifying the conditions that would allow reaching such calls, giving a potential attacker the ability to gain arbitrary code execution.

The amount of source code lines in commonly used C libraries can vary from a few thousand to several hundred thousand. Searching for global function pointers and all the locations where they are called by hand is feasible, but not trivial: manually analyzing a large code base would require a considerable amount of time and effort.

Moreover, even if all function pointers and their calls could be found by hand, the hardest part would still be identifying the conditions over function parameters and other global variables that would lead the program to the execution of such calls. While manually keeping track of all the conditions that need to be satisfied to reach a certain section of the source code at runtime would be feasible, it would be a demanding, time-consuming and error-prone task. Being able to automate this would therefore be a relevant achievement, as it would make the whole process faster, more practical and more reliable.
5. Approach

The goal of Untangle is to provide precise indications about how to reach calls to global function pointers starting from exported functions of a given C library, including the constraints on function parameters and other global variables that need to be satisfied in order to reach such calls. This is achieved through a combination of static analysis, library source code instrumentation, and symbolic execution.

The workflow of Untangle involves two phases. In the build phase, an analyzer component creates a CodeQL database for the library from its source, then performs queries to identify global function pointers, their call sites, and library function that is able to reach them along with their signatures. An instrumenter component then places a call to a uniquely generated target function immediately before each identified global function pointer call, building the instrumented version of the library.

A subsequent symbolic execution phase uses the instrumented binary to evaluate the reachability of identified global function pointer calls, treating these newly inserted functions as targets to reach. An executor component performs the actual symbolic execution through angr with the help of two sub-components: a parser responsible for parsing function signatures and structure definitions, and a memory sub-component implementing Untangle’s custom memory model (using angr’s plugin system) specifically designed to ease the handling of complex pointers to structure parameters.

5.1. Discovery of Global Function Pointer Calls

Once the CodeQL database is built, the analyzer runs a query performing three simultaneous operations: ① detection of all existing global function pointer variables, ② identification of all the call sites for each of the detected variables, ③ discovery of potential entry points to reach the call sites. The last operation involves traversing CodeQL’s call graph, starting from any function containing one or more call sites, going backward from callee to caller and listing all non-static library functions encountered. Whether an identified library function is exported or not can then be checked by looking at the exported symbols of the compiled library binaries.

5.2. Library Instrumentation

Due to its design, CodeQL is unaware of the actual machine code into which the source is compiled. For this reason, binary instrumentation (e.g., of a specific call instruction) is not possible. Therefore, we instrument libraries at the source code level, adding functions whose addresses will then be identifiable through the symbols of the compiled library binaries and used as targets for symbolic execution.

In order to preserve the functionality of the original code while also being able to univocally identify different calls to the same function pointer, the instrumenter creates a unique fictitious target function for each identified call, named TARGET_fptrname_<id>, then wraps every call into a macro named with the same ID that expands to a call to the target function followed by a call to the actual function pointer. Finally, we prevent compiler optimizations from eliminating the fictitious target function call by marking it as noinline and introducing an external side effect in its body (the increment of a global variable, again named with the same ID).

5.3. Symbolic Memory Model

angr’s default memory model [13] is fully symbolic, that is, it emulates any memory operation done by the analyzed program. However, it presents a few shortcomings. Firstly, memory stores concretize symbolic addresses to the maximum permissible value satisfying imposed constraints; secondly, memory loads from symbolic addresses can result in overly complex symbolic expressions or concretize the address to an unpredictable value, chosen at the discretion of the solver. These limitations are a problem when dealing with complex objects that need multiple levels of memory indirection operations to be handled, such as pointers to struct types, since structures are often used to hold other pointers, possibly to other struct types, which in turn could hold even more pointers.

In order to solve this, we override part of the functionality of angr’s memory model, implementing ad-hoc logic to handle nested struct pointers. Function arguments that are pointers to known struct types (extracted through CodeQL) are recursively parsed into an internal StructPointer object, which holds field offsets and sizes, as well as a symbolic bitvector.
During symbolic execution, we keep track of all StructPointer objects, and specially handle load/store memory operations involving symbolic bitvectors belonging to them.

The first load/store operation through the symbolic bitvector of a tracked StructPointer \( p \) will concretize its value to a pre-determined address (incremented each time). At this address, we reserve a chunk of memory of the needed size to hold the contents of the underlying struct that \( p \) is tracking, and then store the symbolic bitvector for any nested StructPointer field of \( p \) at the correct offset in the chunk. The load/store operation to the now-concrete address is then forwarded to angr’s default handler.

![Figure 1: Load/Store handling using Untangle’s memory model](image)

### 5.4. Modes of Operation

Untangle can find constraints on parameters of exported functions and global variables that need to be satisfied to reach identified global function pointer call sites, and then evaluate them to find suitable concrete values.

The parser extracts the number and types of parameters from the signature of each function that needs to be symbolically executed, creating symbolic bitvectors of the appropriate size. For struct pointer parameters, the parser also creates the needed StructPointer objects as previously discussed. Additionally, the executor transforms writable data sections of the library binary to symbolic bitvectors, in order to later verify whether any memory regions belonging to these sections are involved in any constraints, which allows the identification and evaluation of any interesting global variable.

In full library execution mode, the executor selects all target functions previously inserted at instrumentation time as targets to reach. Then it symbolically executes each “interesting” exported function identified through the initial static analysis, and stops at the first target reached.

In filtered execution mode, the same preparation steps are performed, but Untangle lets the user filter call sites, names of exported functions and global function pointers. This mode allows for fine-grained control of the symbolic execution, allowing to select any possible path from identified exported library functions to identified global function pointer call sites, and is useful to explore more options in case full library execution mode reaches a call site that, for any reason, is unacceptable in a specific scenario.

### 6. Experimental Validation

We tested Untangle performing full library execution of several open source C libraries to assess the success rate of symbolic execution and the validity of the found solutions. We implemented a validation mechanism involving the generation and compilation of a small C program that loads the tested library using `libdl` and calls the needed function using parameter values taken from the solution. Untangle automatically creates and executes such a program under a debugger, inserting a breakpoint at the global function pointer call site reached during symbolic execution. If the breakpoint is hit, the call site is indeed also reachable through normal execution using the found solution. The results of these tests can be observed in Table 1.

A solution was found for 877 of the 1488 unique tested functions. Out of these 877, 484 solutions (55%) passed automatic validation. Due to the nature of this approach, false positive validation results are avoided, but false negatives cannot be excluded: exported functions for which no solution was found, but that in reality can still reach a global function pointer call. Identification of false negatives would require prohibitive amounts of manual validation.

Additionally, we also measured Untangle’s performance in terms of used memory (RAM) and execution time. Tests were run on a Debian 11 GNU/Linux v5.10 system using a 64-bit Intel Core i9-10900 CPU, compiling libraries with GCC 10.2.1 with optimizations set to -O2, setting upper limits of 16GiB of RAM and 15
minutes per run. The average memory used per single run was 4.3GiB, and the average execution time was 1m 36s.

Overall, out of 611 failed symbolic execution runs, 70 were instances where the executor did not find a solution, 72 timed out, 291 ran out of memory and 175 failed due to symbolic engine errors.

7. Limitations and Future Work

The main limitations of Untangle come from limitations of CodeQL, which performs its analysis at the source code level and therefore provides no information about the location of instructions or basic blocks in the compiled binaries. Calls that are not performed directly through global function pointer identifiers are not detected by Untangle: their detection would require tracking variable assignments and copies throughout the entire code base. CodeQL offers a mechanism to do this through taint analysis, but is still not able to cover all cases. For instance, if a function pointer is copied into another variable using inline assembly and then the call is performed through the copy, CodeQL does not detect the call, as it doesn’t handle inline assembly.

Another possible limitation is the way Untangle handles code instrumentation: depending on how the target library is written, not every call to a global function pointer found by CodeQL could be actually replaceable with another function call or macro invocation. The reason behind this is that if a function pointer is referenced within a macro definition, CodeQL will detect the source code line where the macro is referenced as the call location. Macros that expand into complex or even partial series of statements cannot be replaced as easily. Being able to extract call site locations from the compiled binary would remove the need to perform source code instrumentation: addresses of call instructions could be directly provided to angr as targets. Frameworks like LLVM, which provide introspection and instrumentation ability at the Intermediate Representation level, could be leveraged to instrument and analyze the generated code.

Finally, because of its design, Untangle needs the source code of the analyzed library: a possible improvement could be the usage of a framework like Joern [14] to perform static analysis of binaries instead, along with heuristics to identify which call sites to consider as global function pointer calls.

8. Conclusions

This work aimed at providing an automated methodology for finding global function pointers whose calls are reachable through exported functions of a C library. The approach we presented employs static source code analysis of a target library to identify global function pointer calls and interesting exported functions, combined with symbolic execution of the latter to find constraints on function parameters and global variables that need to be satisfied in order to reach such calls.

We developed Untangle, a tool that implements this approach with the aim of assisting manual binary exploitation through function pointer hijacking. Untangle relies on an ad-hoc symbolic execution memory model that makes it possible to deal with complex objects, such as pointers to structures, passed as function parameters.

The results we obtained from the tests run on Untangle show that global function pointers can be found in commonly used C libraries and that, under the right conditions, it is possible to reach calls to them starting from exported library functions. Even with the most restrictive state-of-the-art control-flow integrity measures in place, such variables offer a possibility to gain arbitrary code execution if they are overwritten with the address of a carefully chosen legitimate target. Therefore, Untangle provides a reasonable and practical exploitation aid for function pointer hijacking.
References


