Dynamic Text Isolation

Julian S. Levy-Myers
Brown University

Abstract

Many defenses against code-reuse attacks do not protect dynamically shared libraries because instrumentation: (1) occurs in the compilation stage [2] [3], (2) requires debugging information, or (3) depends on accurate binary disassembly [4]. Another obstacle is that software developers do not have access to and/or knowledge of shared libraries configuration on their system. Dynamic Text Isolation (DTI) guarantees only one text segment will be executable in memory during program execution, preventing arbitrary control flow hijacking to code in another code segment. Security policy enforcement occurs when control flow transfers between code segments allowing course-grain control flow integrity (CFI) between dynamic libraries.

1 Introduction

Modern software written in C relies on shared libraries to implement standard library functions and interface with the operating system. Shared libraries such as `glibc`, are a target for attackers because they implement system level functionality such as `evecv` and `system` which can be abused by an attacker to control a system. Just in time return oriented program (JIT-ROP) attacks rely on identifying a sufficient gadget set at runtime to construct a malicious payload. Shared libraries can introduce large amounts of insecure code, significantly increasing an attacker's gadget set. It is difficult for software developers to defend against dynamic code reuse because dynamic library configurations depends on the system and architecture the software is deployed on. For these reasons it is impractical to apply already existing code reuse defenses to shared libraries.

Defenses that protect against code reuse attacks can be placed into two categories. Code randomization attempts to prevent attackers from identifying enough gadgets to carry out an attack. In practice, code randomization is vulnerable to information disclosure, especially from the dynamic linking process. The global offset table (GOT) stores addresses of resolved dynamic symbols. An attacker can read this table and compute the base address of text segment to bypass ASLR. The dynamic linker is another security concern. `ld.so` relies on an ELF data structure for each shared library, which includes information about each code segment's layout in memory. The dynamic linker's data structures can also be read by attackers to disclose dynamic libraries base address as well as symbolic information.

Another approach is CFI, which constructs a control flow graph (CFG) derived from static analysis of a program. Code instrumentation ensures execution only follows edges on the CFG. Even the strictest CFG can can be defeated through bending the control flow [1]. However, CFI can significantly reduce the gadget set of an attacker. Return oriented programming (ROP) has been shown to allow an attacker to do arbitrary computation given a sufficient gadget set [1]. But most attackers do not need arbitrary computation and are targeting a short series of system calls. Enforcing course-grain CFI on library calls can significantly reduce the functionality to which an attacker can redirect execution.

2 Dynamic Text Isolation

Dynamic text isolation enforces three properties at runtime:

1. Only one code segment is executable
2. Execution across code section is restricted to valid dynamic function calls
3. Returns across dynamic libraries are restricted to the call site

Dynamic text isolation intercepts dynamic function calls using the `LD_PRELOAD` flag for `ld.so` dynamic

linker. LD_PRELOAD specifies a shared library which will be first used to resolve dynamic symbols. We instrument the DTI runtime library to have a wrapper function for each valid dynamic function call. When a call is made to a valid dynamic function, the dynamic linker first loads the DTI wrapper function. DTI then calls the dynamic linker a second time to resolve the target function address.

After symbol resolution, DTI marks the code section of the target function executable, and the code section of the call site non-executable. Finally, DTI calls the function. DTI reverse the operation on return, by marking the call site executable, and the call target non-executable. An attacker cannot directly call, jump, or return to a dynamic function because target text segment will be marked non-executable. If an invalid symbol is resolved by the dynamic linker, there is no corresponding DTI wrapper function. The linker will load the target symbol from read only text and fault when the function is called.

2.1 Dynamic Code Remapping

In practice we have have relaxed property 1. The dynamic linker must always be executable because we intercept dynamic symbol resolution. The DTI runtime shared library is always executable, as well as the Linux vsyscall and vdso interfaces.

DTI does not protect program initialization and begins to enforce property 1 when main is first called. We identified libc initialization identifies the address of main, by passing the address to __libc_start_main. The initialization function is a dynamic function which is intercepted by DTI. We replace the address of main with the location of DTI's initialization function. Once libc calls main, DTI marks all code segments that aren't the main executable text, non-executable. Then DTI passes execution to main. On return from main DTI restores all code segments as executable, and returns control to the libc destructor function.

2.2 Identifying the Dynamic Control Flow Graph

We consider the set of valid dynamic functions to be those which are in the ELF dynamic symbol table. Each dynamic symbol used in a program has a corresponding dynamic symbol table entry for the dynamic linker. The ldd utility finds the file path of all shared libraries in the dependency graph of the main executable. DTI instruments wrapper functions for all dynamic function symbols of the main executable, and shared libraries. The set of dynamic function symbols are nodes in the CFG. We allow any call to a valid symbol. Therefore, DTI enforces a CFG with edges between all nodes.

One practical concern when implementing DTI, is intercepting calls made from the DTI runtime shared library. DTI requires mprotect to change memory permissions. If a program also depends on the mprotect library function, DTI will intercept its own call to mprotect. Therefore, DTI does not use any external libraries except the dynamic linker. DTI uses the Linux syscall interface to call mprotect, and mmap. For the same reason, the DTI runtime library must avoid namespace conflicts.

2.3 Restricting function returns

In practice we only restrict function returns, to the address pushed on the stack. In the future, we would like to further restrict function returns to code segments which have a corresponding ELF dynamic symbol entry. This would further reduce the size of the CFG. We can also further restrict returns to call sites, and restrict some dynamic calls to the PLT table.

3 Performance

DTI adds significant overhead to dynamic symbol resolution and dynamic function calls. When a symbol is first resolved to a DTI wrapper function, a second call must be made to dynamic linker to resolve the call target. For subsequent calls, DTI write the address of symbol to memory, so the dynamic linker does not need to be called. Dynamic function calls and returns, require two calls to mprotect to modify the memory permissions.

The instrumentation takes approximately 700 bytes per valid function symbol, as well a small ($<$ 1000 lines of code) DTI runtime library. The executable binary is not modified, and the instrumentation is completely transparent. Loading an additional dynamic library marginally increases program initialization time.

The largest performance cost will be caused by increased instruction cache misses. When DTI changes the memory permissions, the corresponding cache entries will be invalidated. This will significantly increase the runtime of programs that make large numbers of dynamic function calls.

4 Evaluation

This work introduces a transparent method of implementing CFI between code sections. Previous work has focused on hardening static executables. We have shown that code-reuse between dynamic shared libraries presents a significant security threat. DTI significantly

reduces the gadget set of an attacker and restricts control flow across dynamic libraries.

There are several practical concerns that currently make DTI unreliable for commercial software. Code dereferences across dynamic libraries which do not invoke the dynamic linker, result in a segmentation fault. For example, passing a callback function as an argument to a library function will result in a segmentation fault.

DTI does not instrument wrappers for function in libraries which are loaded at runtime. Calling functions from libraries which are loaded after initialization will cause a segmentation fault.

5 Future Work

Future work in the area of securing shared libraries should focus on hardening the dynamic linker. The performance penalty of DTI would be mitigated if security policies were enforced by the linker. There would not be an extra symbol resolution step when a symbol is first resolved. It would also become more practical to track libraries which are loaded at runtime.

DTI can be augmented to allow dereferencing of code pointers passed as arguments. Arguments which are pointers to code can we replaced with a DTI wrapper function. This would allow security policies to be enforced upon dereferencing of dynamic code pointers. We would also like to take advantage of Intel hardware vitalization [2] for quick memory remapping. Sophisticated binary disassembly or compiler assisted techniques would allow DTI to enforce fine-grain CFI [5] policies.

6 Availability

This project is available on github.

`https://github.com/jlevymyers/text-isolation`

References

[1] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 161–176.

[2] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A. R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy* (May 2015), pp. 763–780.

[3] KOO, H., CHEN, Y., LU, L., KEMERLIS, V. P., AND POLYCHRONAKIS, M. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, pp. 472–488.

[4] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 256–267.

[5] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 337–352.