SecGOT: Secure Global Offset Tables in ELF Executables

Chao Zhang, Lei Duan, Tao Wei, Wei Zou
Beijing Key Laboratory of Internet Security Technology
Institute of Computer Science and Technology, Peking University
Beijing, China
{chao.zhang, lei_duan, wei_tao, zou_wei}@pku.edu.cn

Abstract—**Global Offset Table (GOT) is an important feature to support library sharing in Executable and Linkable Format (ELF) applications. The addresses of external modules' global variables and functions are runtime resolved and stored in the GOT and then are used by the program. If attackers tamper with the function pointers in the GOT, they can hijack the program's control flow and execute arbitrary malicious code. Current research pays few attentions on this threat (i.e. GOT hijacking attack). In this paper, we proposed and implemented a protection mechanism SecGOT to randomize the GOT at load time, and thus prevent attackers from guessing the GOT's position and tampering with the function pointers. SecGOT is evaluated against 101 binaries in the */bin* directory for Linux. The results show that it introduced quite low load-time overhead and provides an effective protection against GOT hijacking attacks.**

Keywords-Global Offset Table Hijacking; Function Pointer; Randomization; Dynamic Linker; ELF

I. INTRODUCTION

Library sharing is a common development skill to provide services to independent programs. This skill encourages code and data sharing in a modular fashion, and thus is widely used in modern software developing.

Libraries can be statically linked to or dynamically linked to a main module to generate the final executable binary file. Statically linked libraries will be included in the final executable file after compiling and linking. This will increase the executable file's size and restrain code and data sharing, however.

Dynamic linked libraries are loaded from individual file into memory at load time or run time. And thus their memory addresses are not fixed in order to avoid memory conflicts with other libraries. Code or data in the library which reference absolute memory addresses should be relocated after loading. In addition, the security mechanism ASLR (Address Space Layout Randomization [1]), which are widely adopted by modern operating systems to mitigate attacks like buffer overflows [2], will randomize each module's address at load time. And thus, not only the libraries, but also the main module will be loaded into random addresses at runtime.

Further, if the code in the main module or libraries references an absolute address, it has to be relocated at load time. And thus, the library code cannot be shared between different processes, because the referenced absolute addresses are different in these two processes (i.e. the library code for these two processes are different). This problem also restrains the code sharing feature of libraries. A solution called PIC (Position Independent Code [3]) is proposed for the ELF (Executable and Linkable Format [4]) executable binaries which are common in Linux.

In libraries or main modules supporting PIC, the code section does not reference any absolute addresses in order to support code sharing between processes. However, absolute addresses are unavoidable in programs. As a result, a GOT table (Global Offset Table [4]) is introduced in the library. This table resides in the data section and is not shared between processes. All absolute addresses referenced by the code section are stored in this GOT table. The code section utilizes relative offsets to access these absolute addresses. And thus, the library code can be shared among processes, even if they are loaded into different memory address spaces.

The GOT table may also contain function addresses (i.e. function pointers), such as functions imported from external modules. These function pointers will be called at some time during the program execution. If lazy binding [4] is applied by the linker, this GOT table will be writable during the execution. As a result, if the function pointers in GOT tables are overwritten by attackers, the program's control flow will be hijacked and thus arbitrary code execution is triggered. This kind of attack, i.e. GOT hijacking attack, is first introduced by c0ntex [5].

Researchers pay few attentions on this specific threat. More vulnerable and attractive targets such as return addresses on the stack are the main battlefield between attackers and defenders. Modern operating systems widely deploys security enhancements like ASLR (Address Space Layout Randomization [1]), DEP (Data Execution Prevention [6]) and SafeSEH (Structured Exception Handling [7]) and greatly raises the bar for attackers to exploit these vulnerabilities. However, function pointers are seldom protected and will become the next attack targets.

PointGuard [8] protects function pointers by encrypting and decrypting. It introduces a big runtime overhead and causes compatibility issues. RELRO [9], defines the *LD_BIND_NOW* environment variable to prohibit lazy binding and then sets the GOT to read-only after loading. However, it introduces a big overhead at load time because a lot of unused symbols are resolved at load time.

In this paper, we proposed a protection mechanism SecGOT based on randomization to mitigate GOT hijacking attacks. More specifically, the GOT's relative offset to other sections are randomized at load time, and the entries' order inside the GOT are also randomized. In this way, attackers can hardly guess the target GOT entry's address, and thus cannot overwrite the function pointers stored in the GOT table.

The implementation of SecGOT consists of a plugin of the disassembler IDA Pro [10] and a custom dynamic linker (or the loader in Linux). The plugin recognizes the GOT table and all function pointer entries in this table, and all references to these pointers. Then this plugin stores this information into a custom section in the binary executable file. Our custom dynamic linker reads this section and randomizes the GOT table and its entries at load time, and then updates all references to these entries.

The evaluation of SecGOT against 101 binary executable files shows that SecGOT is efficient and effective. The extra introduced load time is less than 2 milliseconds. Besides, all function pointer entries in GOT are randomized and thus are protected from GOT hijacking attacks.

II. Related Work

GOT hijacking attacks were first introduced by c0ntex [5]. After tampering with the function pointers in GOT, attackers can hijack the control flow and execute arbitrary malicious code.

Previous research focused on protecting function pointers' integrity can mitigate this threat to a certain extent. Methods which aim to defeat stack overflow attacks, such as [11–14], can enforce the integrity of function pointers on the stack. However, function pointers in GOTs are not on the stack, and thus are out of the scope of their protections.

PointGuard [8] is a compiler technique to enforce the integrity of all function pointers by encrypting pointers when stored in memory, and decrypting them only when loaded into CPU registers. It can also protect the function pointers in GOT from tampering. However, it introduces a large overhead, and causes compatibility issues. Even worse, there still are weaknesses [15] in PointGuard. And thus, PointGuard is not deployed in practice.

Other lightweight and efficient protection mechanisms, such as DEP (Data Execution Prevention [6]) and ASLR (Address Space Layout Randomization [1]), are widely adopted in modern operating systems. These protections greatly raise the bar for attackers to hijacking programs' control flow. However, attackers can bypass DEP through overwriting function pointers in GOT with a valid code address, similar as advanced attacks return-to-libc [16] and ROP (Return Oriented Programming [17]). For ASLR, it makes it difficult for attackers to guess target memory's address. However, the current ASLR implementations are limited to randomizations of relatively low granularity, e.g. the relative offset between variables in a module is fixed. And thus, ASLR is not sufficient to protect GOT.

Another straightforward protection against GOT hijacking attacks is to limit the GOT to be read-only. This is done through a load-time adjustment. If the environment variable *LD_BIND_NOW* [9] is defined, the loader (i.e. dynamic linker) will resolve all imported symbols and fill the GOT after loading, and then set the GOT to read-only. However, this will introduce a large overhead when loading because all symbols (even though most of them are not used at all) are resolved at load-time.

III. Background and Threat Model

A. Function Pointers in GOT Table

The GOT table in ELF file stores imported global variables or functions' addresses. For function pointers in the GOT table, they are referenced by the code through a PLT (Procedure Linkage Table [4]).

As shown in Fig.1, if an external function (e.g. *printf*) is called in the code (e.g. step 1 in this figure), the control flow jumps to the PLT's *n-th* entry (e.g. step 2 in the figure). It is worth noting that, the step 2 is done through a relative offset rather than an absolute address of the PLT. The PLT consists of several chunks of code. As shown in this figure, the chunk related to *printf* then jumps to the address stored in the GOT (e.g. the step 3).

Considering the performance, a lazy binding is commonly used when the dynamic linker loads modules. More specifically, the function pointer stored in the GOT table firstly points back to the PLT code chunk (e.g. the step 4 in the figure). So, if *printf* is called for the first time, a resolver of the dynamic linker is called (e.g. the step 5) to retrieve the actual address of the target function. After the resolving, the address of *printf* is written to GOT table (e.g. the step 6) and then *printf* is called. If *printf* is called again in the code, the resolver won't be called again because the address of *printf* is already stored in the GOT. Using this lazy binding, function pointers which are not used at runtime are not resolved. So, it saves a lot of running time.

B. Threat Model

The reference [5] shows a typical GOT hijacking attack. In brief, due to the lazy binding, the GOT table is writable at runtime. And thus, if attackers know where GOT is, they can overwrite GOT entries with the shellcode's address, and

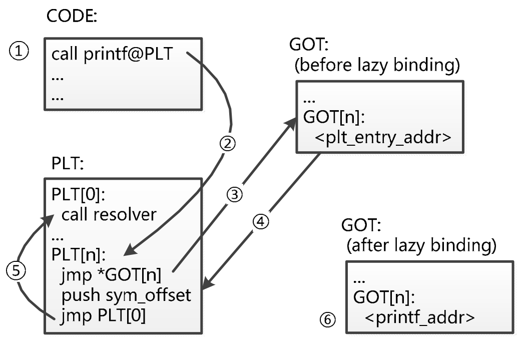

Figure 1. Usage of PLT and GOT

thus hijack the program's control flow. In this paper, we assume attackers have the following abilities or limitations:

- Attackers can access the executable file on the disk. And thus they know the relative offset between sections, functions and so on.

- Attackers may get the address of where the module is actually loaded into memory. And thus, they know all functions' addresses in the module.
- Attackers are not able to read arbitrary memory. For example, they cannot read out the GOT's address through reading the PLT memory.

Under these assumptions, we proposed our protection scheme SecGOT which randomizes the GOT at load time to protect GOT from tampering.

IV. DESIGN AND IMPLEMENTATION

SecGOT consists of two components, one plugin of the disassembler IDA Pro to recognize function pointer entries in the GOT table and all references to them, and one custom dynamic linker which randomizes the executable module or library at load time. Fig. 2 is the architecture of SecGOT.

A. Recognize GOT and Save Infromation

SecGOT first recognizes all GOT entries and their references. An IDA Pro plugin is developed to finish this work. The workflow of this plugin consists of three steps:

- Recognize the GOT table in the target ELF file. The plugin reads the section named "*.got.plt* " to locate the GOT table. This section contains all function pointers of the GOT table.
- Find all references to function pointer entries in the GOT table. For each function pointer entry in the GOT table, all references to this function pointer are identified through the cross-reference information provided by IDA Pro. All these references should be updated if the GOT table is randomized after loading.
- Store all recognized function pointer entries and references into the ELF file. A custom section named "*.secgot*" is defined and inserted to the end of the ELF file. This section consists of blocks related to function pointer entries in the GOT table. More specifically, each block records the address of a function pointer entry in the GOT table and all references to this entry. A custom structure is defined to save this information efficiently.

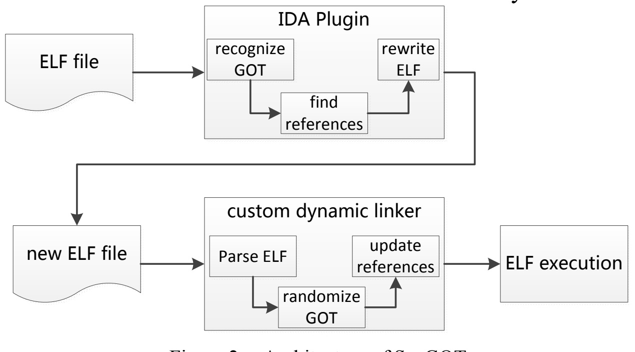

Figure 2. Architecture of SecGOT

All these three steps can be done with the help of the disassembler IDA Pro. The plugin we implemented has about 200 lines of code.

In addition, this plugin also rewrites the section named "*.interp*" in the ELF file. This section defines the absolute path of the dynamic linker which will be invoked when loading the ELF file. We change this path to our custom dynamic linker's path when rewriting the ELF binary.

B. Randomize GOT at Load Time

After the ELF is rewritten by the IDA Pro plugin and launched to execute, a custom dynamic linker is invoked to interpret the new section "*.secgot*" and to randomize the GOT table. It also consists of three steps:

- Parse the ELF file and read out the "*.secgot*" section. Contents in this section are read out, including the GOT table's length, each entry's address and all references' addresses.
- Randomize the GOT table's address and the relative order of all its entries. The pseudo code of this randomization is listed in Table I, i.e. the function *RandomGOT*. It first picks up a random unused memory address for the new GOT table. And then, a record array which keeps the new order of entries is created. A simple randomization algorithm is applied on this record array. Finally, entries in the GOT table are copied into the new GOT table according to this record array, as shown in the function *RandomGOT*.
- Update each reference to the GOT entries. This process is quite simple, as shown in function *UpdateRef* in Table I.

The custom dynamic linker is built upon the existing dynamic linker in the glibc library. In addition to all its existing functionalities, we instrument the previous three steps to the linker. After linking with this new custom linker, the GOT table is randomized and compatible with the code in the ELF file. And attackers can hardly hijack the GOT.

TABLE I. ALGORITHM OF THE GOT RANDOMIZATION AND REFERENCES UPDATING

```
function RandomGOT( got, length ){
    newAddr = rand_unused_mem();  // pick up a random unused memory
    newGOT = new_mem(newAddr, length); // new memory at newAddr
    records = array(length); // record the new order of each entry
    for(int i=0; i<length; i++)
        records[i]=i;      // initialization
    for(int i=0; i< length; i++){
        int j = randInt(0,i);        // random number between 0 and i
        swap(records [i], records [j]);
    }
    for(int i=0; i< length; i++)
        newGOT[ records[i] ] = got[i];  // randomize entries in GOT
    return (newGOT, records);
}

function UpdateRef(got, newGOT, records, length, references){
        for(int i=0; i< length; i++){
            addr = got[i];
            newAddr = & newGOT[ records[i] ];
            for refAddr in references[addr]:
                // update each reference with the new GOT addr.
                update_memory(refAddr, newAddr);
        }
}
```

V. Evaluation

We evaluate SecGOT against the 101 ELF binary executable files in the */bin* directory of the Ubuntu 10.04. All these 101 binaries are rewritten by the IDA Pro plugin first, and then linked with our custom dynamic linker, and finally executed as they are expected.

During this evaluation, we also measure the file size increment, load-time overhead and correctness. These evaluations show that SecGOT is efficient and effective.

- The file size increment after rewriting by the IDA Pro plugin. The result shows that the file size increment is quite small, i.e. all these increments except one are smaller than 2KB.
- The load-time overhead introduced by the custom dynamic linker. Our custom dynamic linker randomizes the GOT before the control transfers to the code section. As a result, an overhead is introduced. The evaluation shows that, the randomization of SecGOT only introduces negligible overhead, less than 2 milliseconds.
- The correctness of the rewritten ELF file. The GOT table is randomized by SecGOT. And thus, if any reference to a GOT entry is missed to be updated, the program will fail to behave correctly. For all these 101 binaries hardened by SecGOT, they are executed for several times. The results show that they behave correctly. It means that SecGOT is compatible with the existing system.

As we can see, the file size increment and the load-time overhead are positively related to the size of the GOT table. We also evaluate the size of the GOT table for each binary.

Table II shows the evaluation of the GOT table size. As shown in the table, all of them have a small GOT table (i.e. less than 200 entries), except for one application (i.e. *edb*). Within the range 41~60, there are 32 applications (including *open* and *kill*). And all these 32 applications' GOT tables' average size is 50.

In addition, the GOT entries are all randomized. And thus, it can protect ELF executables from GOT hijacking attacks.

VI. Conclusion

GOT hijacking attack is an underestimated threat. In this paper, we proposed a scheme SecGOT based on randomization to protect function pointers in GOT tables. Results show that it is an efficient and effective solution.

TABLE II. EVALUATION OF THE GOT TABLE SIZE

GOT size	1~20	21~40	41~60	61~100	101~200	200~
#app	9	15	32	26	18	1
average size	12	31	50	75	141	959
example app	busy-box	kill echo	open sleep	cat ps	cp tar	edb

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. This work is supported by the National Natural Science Foundation of China under the Grant No. 61003216.

References

[1] PaX-Team, “PaX address space layout randomization (ASLR),” 2003. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt.

[2] A. One, “Smashing The Stack For Fun And Profit,” Phrack Magazine, vol. 7, no. 49, 1996.

[3] Yobot, “Position-independent code,” Wikipedia, 2012. [Online]. Available: http://en.wikipedia.org/wiki/Position-independent_code.

[4] System V Application Binary Interface: Intel386 TM Architecture Processor Supplement, 4th ed. The Santa Cruz Operation, Inc., 1997, p. 377.

[5] C0ntex, “How to hijack the Global Offset Table with pointers for root shells.” [Online]. Available: http://www.open-security.org/texts/6.

[6] S. Andersen and V. Abella, “Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies,” MSDN online library, 2004. [Online]. Available: http://technet.microsoft.com/en-us/library/bb457155.aspx.

[7] Microsoft Corporation, “Image has Safe Exception Handlers,” MSDN online library. [Online]. Available: http://msdn.microsoft.com/en-us/library/9a89h429(v=vs.80).aspx.

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “PointGuard TM: Protecting Pointers From Buffer Overflow Vulnerabilities,” in Proceedings of the 12th conference on USENIX Security Symposium, 2003, pp. 91–104.

[9] J. Cohen, “RELRO: RELocation Read-Only,” 2011. [Online]. Available: http://isisblogs.poly.edu/2011/06/01/relro-relocation-read-only/.

[10] Hex-Rays, “IDA Pro: a cross-platform multi-processor disassembler and debugger.” [Online]. Available: http://www.hex-rays.com/products/ida/index.shtml.

[11] M. Frantzen and M. Shuey, “StackGhost: Hardware facilitated stack protection,” Proceedings of the 10th USENIX Security Symposium, 2001.

[12] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, “StackGuard□: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,” Proceedings of the 7th USENIX Security Symposium, 1998.

[13] S. Sinnadurai, Q. Zhao, and W. Wong, “Transparent Runtime Shadow Stack□: Protection against malicious return address modifications,” pp. 1–11.

[14] N. Tuck, B. Calder, and G. Varghese, “Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow,” 37th International Symposium on Microarchitecture (MICRO-37’04), pp. 209–220, 2004.

[15] S. Alexander, “Defeating compiler-level buffer overflow protection,” USENIX;LOGIN, vol. 30, no. 3, pp. 59–71, 2005.

[16] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” Proceedings of the 11th ACM conference on Computer and communications security (CCS’04), p. 298, 2004.

[17] E. Buchanan, R. Roemer, and H. Shacham, “When good instructions go bad: generalizing return-oriented programming to RISC,” Proceedings of the 15th ACM conference on Computer and communications security, pp. 27–38, 2008.