Squeezing the Dynamic Loader For Fun And Profit

Mingwei Zhang R. Sekar
Stony Brook University
{mizhang,sekar}@cs.stonybrook.edu

Abstract

Dynamic loader is a powerful code module that has privileges to support debugging, profiling and program loading. Its privilege includes the power to map executable memory region; unprotect code/data regions to patch relocations. Due to these privileges and its wide deployment, the effect of bugs and vulnerabilities in dynamic loader is devastating.

In this work, we demonstrate a code injection attack in dynamic loader is feasible. We corrupt the loader data structure and transfer the control to an internal loader function. Our experiment shows that we are able to change existing write protected code to our shellcode. We show that the attack is easy to achieve and requires a very simple ROP gadget chain.

Keywords Dynamic Loader, Code Injection, Return-Oriented Programming

1. Introduction

Software nowadays is usually deployed in form of dynamic linked binaries including executables as well as dependent libraries. dynamic linked binaries enjoy the benefits of the convenience to be updated, modularity, flexibility, easiness for code sharing and etc. Thanks to these benefits, dynamic linked link binaries becomes the de-facto choice for software deployment.

Dynamic loader place a key role in the dynamic linking process. It is responsible for resolving dependencies by loading library files and managing all code modules in the runtime. Therefore, Dynamic loader is a powerful code module that has lots of privileges. In particular, its privilege includes the power to map executable memory region for loading modules; unprotect code/data regions to patch relocations. Due to these privileges and its wide deployment, the effect of bugs and vulnerabilities in dynamic loader is devastating. On the other hand, privileges of dynamic linker gives attackers a huge capability to bypass existing security defense and increase the possibility of the code injection attack.

Our research discovers that attackers could easily reuse these privileges. In particular, we demonstrate a code injection attack in dynamic loader. This is done by corrupting the loader data structure and transferring control to an internal loader function. Our experiment shows that we are able to change existing write protected code to our shellcode. The attack is easy to achieve and requires a very simple ROP gadget chain.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Existing Loader Security Hardening

2.1 Relocation Read Only

Relocation read only (short for RELRO) is security hardening mechanism that protects function pointers and important metadata in ELF binaries from corruption attacks. These functions or metadata usually requires dynamic loader to perform relocation (patch values) to cope with ASLR. RELRO makes sure these important data is marked read-only after relocation.

To fully protect all these metadata, dynamic loader need to patch all of them, which is often time consuming. For instance, resolving an external function address in Global Offset Table (GOT) often requires numbers of string match on different function names and hash checking in different libraries.

To make a trade off between performance and security, developers of glibc and gcc designed two types of RELRO: partial-RELRO and full-RELRO. Among the metadata to protected, there are internal function addresses, dynamic segment, loader metadata (GOT[1] and GOT[2]), and imported function addresses.

partial-RELRO protects all of them except the last one: imported function addresses, which full-RELRO also protects. Clearly, full-RELRO requires eager symbol binding which is more time consuming.

2.2 Caller Address Checking

To prevent unauthorized modules invoking internal functions of dynamic loader, developers of glibc add some security checking on their critical unexported functions. In par-

ticular, they checks the return address of the caller and see if they fall into allowed modules. For instance, a inline check `__check_caller` is put in some internal critical functions such as `dl_open_worker`.

3. Code Injection Using Text Relocation

Note that code sections are normally write protected, thus preventing attacks that overwrite them. Text relocation is a convenient key to open the "door." Although text relocation is discouraged since it makes it difficult to share code memory across processes, it is still used in some libraries and its code is available in all versions of glibc loader.

This section describes how to launch a code injection attack by leveraging the loader code and data using text relocation. The attack is launched simply in the following steps:

- Bypass ASLR and figure out loader data structure
- Corrupt loader data structure
- Transfer control to loader function

3.1 Bypass ASLR

The first step is to by pass ASLR and figure out loader data structure and text relocation function inside loader. Dynamic loader contains a data structure called "link_map" for each code module. This data structure stores the metadata information such as the address of symbol table, address of relocation table and whether the module has been processed, as shown in Figure 1.

There are several methods to figure out the memory location of `link_map`. The first one is to check the Global Offset Table (GOT). In fact the 2nd element of GOT (GOT[1]) contains the address of `link_map` in most of binaries compiled with partial RELRO. In case for binaries that are compiled with full RELRO or binaries launched with eager binding (with parameter `LD_BIND_NOW`, GOT[1] is not initialized.

However, none of these counter measures stop an attacker. Our experiment shows that in program binary, there is an special memory segment called `.dynamic`, which contains information of the binary at runtime. In particular, there is an entry called `DT_DEBUG`. This entry points to a data structure that contains a pointer to the address of `link_map`.

Although it is easy for dynamic loader to disable `DT_DEBUG` by simply eliminating 2 lines of its source code, we argue that such code modification will be unlikely, since it poses inconvenience for program debugging.

Once `link_map` address is known, it is easy to figure out text relocation function address. This is because `link_map` is stored as an array, while the 1st one is for the executable and one after is for the dynamic loader. From the 2nd `link_map` data structure, we will know the base address of loader. Then we could use binary scanning to find out the function address.

```
struct link_map
{
  ElfW(Addr) l_addr;  /* Base address  */
  ElfW(Dyn) *l_ld;    /* Dynamic section */

  ElfW(Dyn) *l_info[DT_NUM];

  /* l_info[DT_TEXTREL] = 1
   * l_info[DT_REL] = attacker_rel;
   * l_info[DT_SYMBL] = attacker_sym;
   * l_info[DT_GOTPLT] = attacker_got;
   */

  const ElfW(Phdr) *l_phdr;  /* program header*/
  unsigned int l_relocated:1;
};
```

Figure 1. Dynamic Loader Internal Data Structure

3.2 Corrupt Loader Data Structure

When memory address of "link_map" is leaked, we leverage a memory corruption attack to corrupt data pointers of relocation table and symbol table to point to our own payload. In addition, we modify a flag in link_map for text relocation patching. Specifically, this flags indicates loader to unprotect write protected code pages. and this flag is corrupted into value "DT_TEXTREL" as shown in Figure 1. The memory address specified by the crafted relocation will be updated and by the value specified by the crafted symbol table. This way attack could write to arbitrary code location with any value.

The link_map data structure shown in figure 1 contains related metadata for this attack, such as , pointers for relocation table (l_info[DT_REL]), string table (l_info[DT_STR]), symbol table (l_info[DT_SYMBL]) and etc. These pointers do not directly point to their data but all point to the locations inside the read-only dynamic section (when RELRO is applied). This does not prevent attacker, since they could corrupt the data pointer to attackers' payload. In addition, to launch a successful attack, l_relocated is a flag that should be flipped, since it indicates whether the object has been relocated. Finally, GOT needs to be taken over too, because, the function we are using will initialize GOT table. However, after relocation patching, the .got section in ELF file become read-only, this will trigger a segfault. Changing the GOT to any arbitrary location that contains twelve bytes of writable memory should work (3 GOT entries).

3.3 Invoke Text Relocation Function

When the link_map data structure is corrupted, code injection attack can be launched by invoking an internal function (`_dl_relocate_object`).

(`_dl_relocate_object`) takes 4 parameters, the first parameter is the address of `link_map`. The 2nd one is not used.

The third one is the relocation mode, here, the value can be simply 0x1 (representing `RTLD_LAZY`). And the last one indicates whether doing profiling or not. Again, pass an integer 0 is fine.

To simplify our prototype, we use a simple program that contains a buffer overflow which allows attacker to change the content of stack including return addresses. Using this exploit, we could easily launch the text relocation function in one shot. Further, we modify the stack frame pointer to make sure when function returns, it goes to our injected payload.

This runtime relocation attack used in above case can only corrupt the current module of the link_map data structure. However, it can be generalized to corrupting all memory region in the runtime. This is true if attackers work harder to corrupt one more pointer, l_phdr. l_phdr points to the ELF program header table located in the write protected ELF image. Corrupting this data pointer allows attacker to fool dynamic loader to unprotect and corrupt arbitrary code or data.

4. Related Work

Despite the spate of vulnerabilities and bugs [1–7] in dynamic loader, the loader security has not drawn much attention until recently. Kwon et. al. proposed their work on automatic detection of unsafe module loading [8]. Their work is to solve the resolution hijacking issue by using dynamic instrumentation and profiling. Our attack cannot be detected by their approach, because it works in a lower level and modify the code of a legitimate module loaded from correct path.

The idea of securing the loader was first presented by M. Payer in his work, TRuE [9], a dynamic binary translation system (DBI). The paper solves two problems: 1) protecting dynamic loader internal data structures. 2) make sure DBI not subverted at program startup. Their approach is to implement a secure loader to make sure their DBI system get loaded first, and then DBI system write protect all loader data structures.

After careful evaluation of the source code of the secure loader, we find that there are still chances that our attack could succeed. This is because, 1) some APIs does not have sufficient parameter checking, which allows us bypass their defense. 2) the code cache in their DBI system is writable and executable. 3) TOCTOU attack still allows us to modify loader data structures during loading period.

There exist other low level attacks using loader's code. The "weird machine" [10] proposed by Shapiro et. al. leverages the code of loader processing relocation to craft a Turing complete machine. They achieve this by craft their own relocation table. Similar to our attack by using relocation, their attack is more of proof-of-concept, since huge amount of relocation is required, which is infeasible in exploit environment.

5. Conclusions

References

[1] CVE-2000-0854: Earliest side-loading attack.

[2] CVE-2010-3847: privilege escalation in loader with $origin for the ld_audit environment variable. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3847.

[3] CVE-2010-3856: privilege escalation in loader with the ld_audit environment. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3856.

[4] CVE-2011-0562: Untrusted search path vulnerability in adobe reader.

[5] CVE-2011-0570: Untrusted search path vulnerability in adobe reader. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0570.

[6] CVE-2012-0158: Side loading attack via microsoft office. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0158.

[7] CVE-2013-0977: overlapping segments. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0977.

[8] T. Kwon and Z. Su. Automatic detection of unsafe component loadings. In *the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 107–118, New York, NY, USA, 2010. ACM.

[9] M. Payer, T. Hartmann, and T. R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *S&P*, 2012.

[10] R. Shapiro, S. Bratus, and S. W. Smith. "weird machines" in elf: A spotlight on the underappreciated metadata. In *the 7th USENIX Conference on Offensive Technologies*, WOOT'13, pages 11–11, Berkeley, CA, USA, 2013. USENIX Association.