RED HAT BLOG

Hardening ELF binaries using Relocation Read-Only (RELRO)

January 28, 2019 | Huzaifa Sidhpurwala

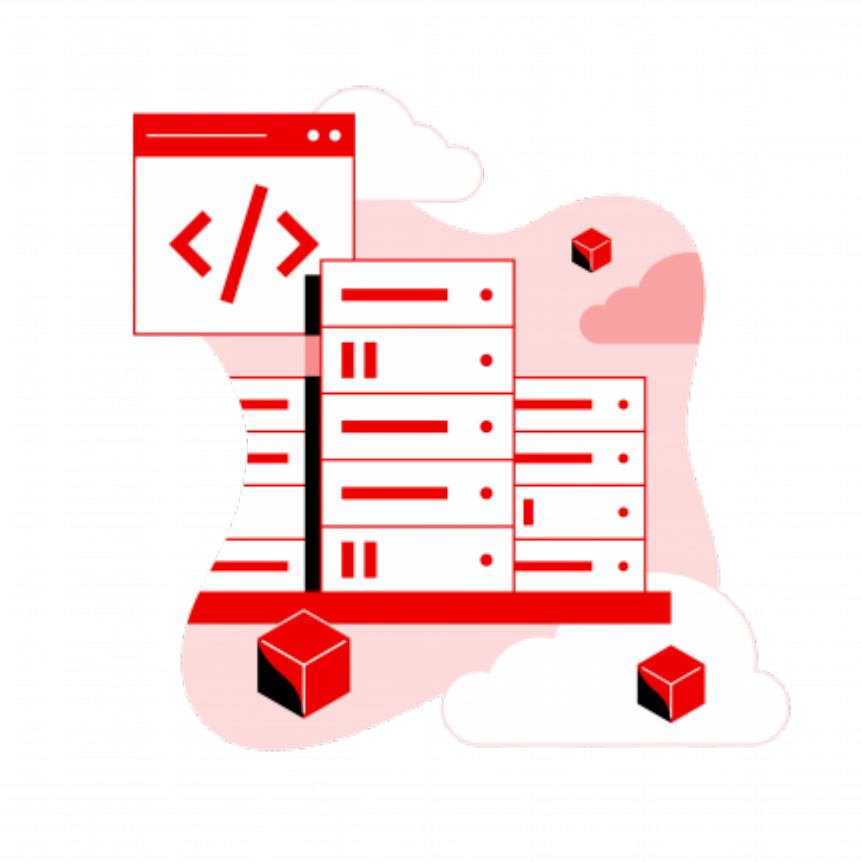
< Back to all posts

Tags: *Security*

Linux uses the Executable and Linkable Format (ELF) as a common standard for executables (binaries), object code, shared libraries, and even core dump. ELF is flexible, extendable and cross-platform.

One mechanism to harden ELF binaries is to use Position Independent Executables (PIE) that are an output of the hardened package build process. A PIE binary and all of its dependencies are loaded into randomized locations within virtual memory each time the application is executed. This makes Return Oriented Programming (ROP) attacks much more difficult to execute reliably.

Several other techniques exist for hardening ELF binaries in Linux. This post discusses one such technique called Relocation Read-Only (RELRO). Selected network daemons and suid-root programs on Red Hat Enterprise Linux version 7 (on architectures which support RELRO) are built with RELRO support. All ELF binaries shipped with Fedora version 23 and later are built with full RELRO support.



Read more about optimizing performance for the open-hybrid enterprise.

Visit our Red Hat Enterprise Linux (RHEL) Performance Series page →

A dynamically linked ELF binary uses a look-up table called the Global Offset Table (GOT) to dynamically resolve functions that are located in shared libraries. Such calls point to the Procedure Linkage Table (PLT), which is present in the .plt section of the binary. The .plt section contains x86 instructions that point directly to the GOT, which lives in the .got.plt section. GOT normally contains pointers that point to the actual location of these functions in the shared libraries in memory.

The GOT is populated dynamically as the program is running. The first time a shared function is called, the GOT contains a pointer back to the PLT, where the dynamic linker is called to find the actual location of the function in question. The location found is then written to the GOT. The second time a function is called, the GOT contains the known location of the function. This is called “lazy binding.” This is because it is unlikely that the location of the shared function has changed and it saves some CPU cycles as well.

There are a few implications of the above. Firstly, PLT needs to be located at a fixed offset from the .text section. Secondly, since GOT contains data used by different parts of the program directly, it needs to be allocated at a known static address in memory. Lastly, and more importantly, because the GOT is lazily bound it needs to be writable.

Since GOT exists at a predefined place in memory, a program that contains a vulnerability allowing an attacker to write 4 bytes at a controlled place in memory (such as some integer overflows leading to out-of-bounds write), may be exploited to allow arbitrary code execution.

RELRO

To prevent the above mentioned security weakness, we need to ensure that the linker resolves all dynamically linked functions at the beginning of the execution, and then makes the GOT read-only. This technique is called RELRO and ensures that the GOT cannot be overwritten in vulnerable ELF binaries.

RELRO can be turned on when compiling a program by using the following

It's also possible to compile with partial RELRO, which can be achieved by using the "`- z,relro` " option and not the " `–z,now` " on the gcc command line.

In partial RELRO, the non-PLT part of the GOT section (.got from readelf output) is read only but .got.plt is still writeable. Whereas in complete RELRO, the entire GOT (.got and .got.plt both) is marked as read-only.

Both partial and full RELRO reorder the ELF internal data sections to protect them from being overwritten in the event of a buffer-overflow, but only full RELRO mitigates the above mentioned popular technique of overwriting the GOT entry to get control of program execution.

Example of exploit mitigation when using RELRO

Using Partial RELRO:

Let us consider the following example code:

```
#include <stdio.h>
#include <stdlib.h>


int main(int argc, int *argv[])
{
size_t *p = (size_t *) strtol(argv[1], NULL, 16);

p[0] = 0xDEADBEEF;

printf("RELRO: %p\n", p);

return 0;
}
```

The above code attempts to write the hex constant, 0xdeadbeef, at an address provided by the user on the command line. This is an oversimplified form of a buffer overflow, where an attacker has control over the overwrite address.

Compile this code with partial RELRO:

```
gcc -g -Wl,-z,relro -o test testcase.c
```

Check the binary using checksec (this package is available in Fedora and EPEL):

```
$ checksec -f test
[*] '/home/huzaifas/test'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Let's check where the GOT entry for printf is located using objdump:

```
$ objdump -R test | grep -i printf
0000000000601018 R_X86_64_JUMP_SLOT  printf@GLIBC_2.2.5
```

Now, let's run the code via gdb:

```
$ gdb -q ./test
Reading symbols from ./test...done.
(gdb) run 0000000000601018
Starting program: /home/huzaifas/test 0000000000601018
```

```
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.25-13.fc2

Program received signal SIGSEGV, Segmentation fault.
0x00000000deadbeef in ?? ()
(gdb)
```

We can see that the GOT is overwritten such that printf now points to attacker-controlled address. When printf is called, arbitrary code is executed. This demonstrates how partial RELRO is not really effective in preventing GOT overwrite attacks.

Using full RELRO

Now let's compile the same program using full RELRO as follows:

```
$ gcc -g -Wl,-z,relro,-z,now -o test testcase.c
```

Check if full RELRO was enabled:

```
[huzaifas@babylon ~]$ checksec test
[*] '/home/huzaifas/test'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Let's check where the GOT entry for printf is:

```
$ objdump -R test | grep -i printf
0000000000600fe0 R_X86_64_GLOB_DAT  printf@GLIBC_2.2.5
```

Notice how the GOT entry changed from R_X86_64_JUMP_SLOT to R_X86_64_GLOB_DAT when full RELRO was used.

Run the program using gdb:

```
$ gdb -q ./test
Reading symbols from ./test...done.
(gdb) run 0000000000600fe0
Starting program: /home/huzaifas/test 0000000000600fe0
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.25-13.fc2

Program received signal SIGSEGV, Segmentation fault.
```

```
0x0000000000400550 in main (argc=2, argv=0x7fffffffdd98) at testcase.c:8
8 p[0] = 0xDEADBEEF;
```

The application crashes with SIGSEGV when an attempt is made to overwrite the GOT. This is because GOT is marked as readonly, thereby mitigating the exploit.

In conclusion, RELRO is a generic mitigation technique to harden the data sections of an ELF binary/process. Using full RELRO has a slight performance impact during application startup (as the linker has to populate the GOT entries before entering the main function). Also though RELRO does not require special CPU support (beyond an MMU), toolchain support is needed (particularly from the link editor) before a platform can support it.

ABOUT THE AUTHOR

Huzaifa Sidhpurwala
Principal Product Security Engineer

Huzaifa Sidhpurwala is a Principal Product Security Engineer with Red Hat and part of a number of upstream security groups such as Mozilla, LibreOffice, Python, PHP and others. He speaks about security issues at open source conferences, and has been a Fedora contributor for more than 10 years.

Read full bio →

SUBSCRIBE TO THE FEED

Related posts

A Brief History of Cryptography

Protecting your intellectual property and AI models using Confidential Containers

Ask An OpenShift Admin episode 117: Security considerations while designing a CI/CD Pipeline

Products

Tools

Try, buy, & sell

Communicate

About Red Hat

We're the world's leading provider of enterprise open source solutions—including Linux, cloud, container, and Kubernetes. We deliver hardened solutions that make it easier for enterprises to work across platforms and environments, from the core datacenter to the network edge.

Select a language

English

About Red Hat
Jobs
Events
Locations
Contact Red Hat
Red Hat Blog
Diversity, equity, and inclusion
Cool Stuff Store
Red Hat Summit