Simple Polymorphic x86_64 Runtime Code Segment Cryptor
These are not the bytes you are looking for

GLENN MCGUIRE *(z5012384)*
Extended Digital Forensics (COMP6445)
October 14, 2017

Background

The purpose of code-segment encryption is to hide the behaviour of an executable binary. It makes it such that one must generally execute the binary to discover what the executable actually does - as static analysis of the file is usually infeasible due to complex encryption of the code section.

A polymorphic cryptor is one that changes the encryption key each time the binary is executed, and this is generally done by modifying the executable file during the process startup. This can be achieved by changing the entry point in the executable header to a stub that is inserted within the executable somewhere, usually in a sufficient length of null bytes, which performs the required syscalls and memory operations to modify the executable file on disk, such that the encryption key is different each time the executable is ran.

The benefit of a polymorphic cryptor is that each time the executable is ran, the hash and thus the general signature of the executable file changes - thus making it difficult for antivirus engines that rely on signature checking to detect the malicious software.

Runtime Decryption

To help explain how the polymorphic runtime cryptor works, the *strace* of a basic hello world program that has been packed with the cryptor will be shown to detail the process during execution.

```
$ strace ./hello.packed > /dev/null
execve("./hello.packed", ["./hello.packed"], [/* 37 vars */]) = 0
open("./hello.packed", O_RDONLY)          = 3
mmap(NULL, 912704, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8eea3b1000
unlink("/tmp/ab")                         = -1 ENOENT (No such file or
    directory)
open("/tmp/ab", O_RDWR|O_CREAT, 0777)     = 4
ftruncate(4, 912704)                      = 0
mmap(NULL, 912704, PROT_READ|PROT_WRITE, MAP_SHARED, 4, 0) = 0x7f8eea2d2000
munmap(0x7f8eea3b1000, 912704)            = 0
```

```
close(3)                              = 0
unlink("./hello.packed")              = 0
rename("/tmp/ab", "./hello.packed")   = 0
getrandom("\277", 1, 0)               = 1
munmap(0x7f8eea2d2000, 912704)        = 0
close(4)                              = 0
mprotect(0x400000, 648724, PROT_READ|PROT_WRITE|PROT_EXEC) = 0
mprotect(0x400000, 648724, PROT_READ|PROT_EXEC) = 0
uname({sysname="Linux", nodename="ubuntu-xenial", ...}) = 0
brk(NULL)                             = 0x187a000
brk(0x187b1c0)                        = 0x187b1c0
arch_prctl(ARCH_SET_FS, 0x187a880)    = 0
readlink("/proc/self/exe", "/home/ubuntu/elf/hello.packed (d"..., 4096) =
    39
brk(0x189c1c0)                        = 0x189c1c0
brk(0x189d000)                        = 0x189d000
access("/etc/ld.so.nohwcap", F_OK)    = -1 ENOENT (No such file or
    directory)
fstat(1, {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 3), ...}) = 0
ioctl(1, TCGETS, 0x7ffd36b95e60)      = -1 ENOTTY (Inappropriate ioctl
    for device)
write(1, "hello world\n", 12)         = 12
exit_group(0)                         = ?
+++ exited with 0 +++
```

As can be seen in the output, the binary being executed is opened as *read only* and assigned a file descriptor, after which a new memory mapping is created based on the file descriptor. The temporary file is deleted if it exists, and then is opened and mapped the same way the executing binary was, however this time as writeable.

It is interesting to note that the executable file corresponding to the current executing process cannot be opened in writeable mode. This was the cause of a lot of issues when writing the polymorphic cryptor. The error code $ETXTBSY$ is returned when this is attempted to be done.

#define ETXTBSY 26 / Text file busy */*

After the temporary file is mapped into the packed binary's address space, *ftruncate* is called on the file descriptor to ensure the file is made the same size as the executable running. Although not shown in the output of strace, the startup assembly, or *stub*, inserted into the binary then copies all data from the current executable file mapping to that of the new temporary file mapping, thus copying the file contents completely.

When completed, the stub closes the file descriptor for the executable of the current process, as it is no longer needed. It is also unmapped from the process address space, and deleted from the filesystem with the *unlink* syscall. It is interesting to note that since the executable is still running, the inodes corresponding to the file in the filesystem are not actually deleted - only the directory entry is until which point there are no longer any more references to the corresponding inodes - which will most likely be when the process ends.

The temporary file is then renamed to have the same path and filename as the executable binary that was just unlinked with the *rename* syscall, thus completing the replication process to be able to write to the binary without triggering the **ETXTBSY** error code. Since the executable file is still mapped into the process address space, the xor key can be pulled from further down in the stub code and used to decrypt the *.text* section of the executable binary mapping. Once completed, the xor key in the decryption assembly within the file mapping can be changed, and then the *.text* section within the mapping can be encrypted using this new xor key. The xor key is randomized each time by using the *getrandom* syscall.

The polymorphic encryption of the executable has now been completed, and the stub can now mprotect the *.text* section of the current process, making it writeable so that it may loop over the *.text* section and decrypt it with the current xor key - the same that was used to decrypt the file mapping. Upon finishing, the *.text* section is then mprotected such that it is no longer writeable - otherwise this would raise red flags - before then jumping to the original entry point of the binary.

Every execution of the executable will do the same thing, randomizing the xor key and re-encrypting the *.text* section of the executable file mapping before then decrypting the current process.

Example Output

The following is an example of the polymorphic runtime cryptor in use. A basic hello world program was compiled as a static binary on a x86_64 machine, and packed with the cryptor. It was then executed in a loop, printing the sha256 hash upon each execution.

```
$ for i in {0..5}
do
sha256sum ./hello.packed
./hello.packed
done

f409efebc5bb7bef7bbe256e42861447cba03c446a83ecf03f750bcba86c8d67  ./hello.packed
hello world
7b9db773b0b61637a06d9fab3d4f4fd3e755a0c36b42905e4c75f9acbc2a18c7  ./hello.packed
hello world
afc2c54f4ebfa493f37443642a7898e77e04ee5d6944b0795608ed99003e51bc  ./hello.packed
hello world
0975379c76da4d3a83b0783dfd27a54a80949c138855b82162d4b10558c86996  ./hello.packed
hello world
0714c9265a880312ce785c51d6a00d13e58bb3e362d3d0cdee13d4d9c1ca19f7  ./hello.packed
hello world
c8a1543ab79f93f7052d9531666ae6a93b80d4987a3a32aea01bb43823826b89  ./hello.packed
hello world
```

As can be seen, upon each execution the hash of the binary is different from the last time it was executed. This demonstrates the runtime polymorphism of the stub.