RHEL Crypto Team

Checking your cryptography usage with eBPF

Simo Sorce
Sr. Princ. Sw. Engineer

Red Hat

Auditing cryptography usage

Cryptography is implemented in applications mostly through libraries dynamically loaded from the system

- GOOD: we have a few, well managed and maintained cryptography providers
- BAD: applications do not provide uniform access to information about cryptography use, nor libraries can surreptitiously open files to log data

Auditing cryptography usage

How do we learn what our system is actually using in daily operations?

How do we make sure that the configurations we set are being actually honored?

How do we gather statistics to inform our future decisions?

Tracing as a form of auditing

A similar problem has been solved previously: Performance profiling

Tools:

- Tracing via debug statements
  - Generally not ok for production
- Tracing via user space tools ptrace, gdb
  - Also not ok for production
- Tracing via eBPF
  - Low impact, usable in production

What is eBPF?

Extended Berkeley Packet Filter

- BPF was initially used just for packet filtering, hence the name
- It is a limited Virtual Machine (with optional JIT) running “arbitrary code” in kernel
- Can intercept other code running in the kernel and in user space and perform additional computations defined by programs loaded dynamically in the kernel
- Requires root privileges in most cases

Why eBPF ?

Allows to intercept any function in **any** library loaded in the system

Allows to gather data in tables that can be later queried by a user space program

Generally low performance impact

Does **not*** require changes to the code under inspection

- Although probes in the code makes it more usable

What to monitor?

One thing I wanted to monitor is what TLS ciphers are actually negotiated by my machine.

Until TLS 1.2 there are a gazillion ciphers that can be used, do I really need to enable them all?

Let’s try to find out.

Probing our system (uprobes)

```
#!/usr/bin/bpftrace

BEGIN
{
        printf("Tracing selected ciphers... Hit Ctrl-C to
end.\n");
        printf("%-6s %s\n", "PID", "CIPHER");
}

uretprobe:/lib64/libssl.so.1.1:ssl3_choose_cipher
{
        printf("%-6d %lx,%s\n", pid, *retval,
str(*(retval+8)));
}

uprobe:/usr/lib64/libssl3.so:ssl_ClientSetCipherSuite
{
     @start[tid] = nsecs;
     @suite_counter[arg2] = count();
     printf("%-6d [%x]\n", pid, arg2);
}
```

Probing our system

Using uprobes

- Pros:
  - Easy to set up quickly
  - Can be done even with a one liner from your shell for simple things
- Cons:
  - Requires all debuginfo packages installed
  - Somewhat hard to pull data from complex data structures
  - Might need probe adjustment when library internals change

Instrumenting our system

Second try, USDT probes

```
diff -up nss/lib/ssl/ssl3con.c.usdt nss/lib/ssl/ssl3con.c
--- nss/lib/ssl/ssl3con.c.usdt  2020-01-03 15:27:43.000000000 -0500
+++ nss/lib/ssl/ssl3con.c       2020-01-15 14:37:49.607416077 -0500
@@ -35,3 +35,4 @@

 #include <stdio.h>
+#include <sys/sdt.h>

@@ -6644,3 +6645,7 @@

     ss->ssl3.hs.cipher_suite = (ssl3CipherSuite)suite;
+
+    /* Add USDT probe to report the selected cipher for the connection */
+    DTRACE_PROBE1(cryptoaudit, nss-tls-cipher, ss->ssl3.hs.cipher_suite);
+
     return ssl3_SetupCipherSuite(ss, initHashes);
```

Instrumenting our system

Using USDT (User Statically-Defined Tracing) probes

- Pros:
  - Easy to get just the data you want
  - No debug packages involved
  - No need to adjust probing code over time*
- Cons:
  - Requires new builds with source level changes

How to gather data

There a few ways to enable probes and source data from them

- Bpftrace tool
- BCC (BPF Compiler Collection)
  - Ready made BCC Tools
  - Custom C/C++ programs
  - Custom Python programs using BCC bindings
    - I chose this!

Example: add a probe with python

BPF can be easily accessed by python programs, this makes iterating and experimenting very easy

- Just do:

```
$ python3
>>> from bcc import BPF
>>> help(BPF)
```

The actual BPF code

We need to create a probe first:

```
openssl_tls_cipher_probe_text = """
    #include <uapi/linux/ptrace.h>

    struct cipher_key_t {
        u32 cipher;
    };
    BPF_HASH(ciphers, struct cipher_key_t);

    int count_cipher_use(struct pt_regs *ctx)
    {
        struct cipher_key_t key = {};

        bpf_usdt_readarg(1, ctx, &key.cipher);
        ciphers.increment(key);

        return 0;
    }
"""
```

Structure used to pass data to user space

Hashmap, counts each cipher's invocations

Code that will be executed in kernel each time the probe is triggered

Installing the probe

Next we need to install it in the kernel:

```
# load BPF program
u = USDT(path='/lib64/libssl.so.1.1')

u.enable_probe(probe="openssl-tls-cipher", fn_name="count_cipher_use")

B =  BPF(text=openssl_tls_cipher_probe_text, usdt_contexts=[u])
```

Dynamic Library to intercept

Name of the USDT probe

Name of the function to attach to the probe

BPF program previously defined

Collect data

Finally:

```
while True:
    try:
        #print data every 5 seconds
        time.sleep(5)
        c = b["ciphers"]
        for k, v in sorted(c.items(), key=lambda c: c[1].value):
            print("{}: {}".format(k.cipher, v.value))
    except KeyboardInterrupt:
        exit()
```

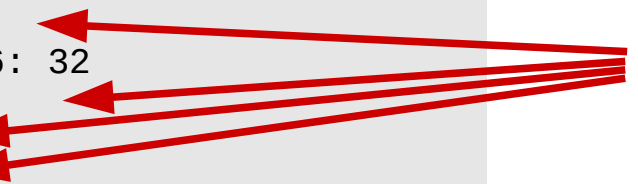
Hashmap used to pass data to user space

Prints cipher number and count

Output example

```
OpenSSL TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384: 1
OpenSSL TLS_AES_128_GCM_SHA256: 2
OpenSSL TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256: 32
OpenSSL TLS_AES_256_GCM_SHA384: 79
NSS TLS_AES_256_GCM_SHA384: 171
NSS TLS_AES_128_GCM_SHA256: 207
NSS TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256: 525
```

Firefox



Data:

457 connections used TLS 1.3
556 connections used TLS 1.2

251 connections used 256bit security
766 connections used 128bit security

Findings: Firefox reconnects a lot and the Web I use is mostly TLSv1.2 - 128bit

Thank you

Questions?

linkedin.com/company/red-hat

facebook.com/redhatinc

youtube.com/user/RedHatVideos

twitter.com/RedHat

Red Hat