Reverse Engineering the World's Largest Music Streaming Platform

@devgianlu - FOSDEM 2026

Who am I?

- Backend Software Developer
- Open Source enthusiast and contributor
- Software Security and Reverse Engineering enthusiast
- Hobbyist IoT researcher
  - CVE-2023-3741 and CVE-2024-3016
  - More fun with other stuff
- CTF player
  - Former TeamItaly member
  - ECSC 2024 organizer @ CybersecNatLab
  - ICC 2023 provider @ CybersecNatLab
  - DEFCON finalist w/ mhackeroni

The mandatory disclaimer ⚠️

This talk expresses **my opinion**, not the one of my current, previous or future employer, not of my friends, colleagues or people I met online.

All the knowledge presented in this talk was **acquired by me personally**, work done by others has been intentionally left out.

All the work done here aims to **enhance Spotify**, not undermine it.

Please don’t use this work to do piracy, I don’t have anything to do with Anna’s Archive dump and won’t ever participate in this kind of activities.

To Spotify, please don’t ban me again and please don’t sue me, let’s have a chat first. This talk was scheduled way before the recent events.

What is librespot?

librespot is a client library for Spotify, capable of playback via various backends.

The project is:

- An headless player
- An implementation based on reverse-engineering
- A fully* capable Spotify Connect endpoint
- An alternative to old-time deprecated *libspotify*

The project is **not**:

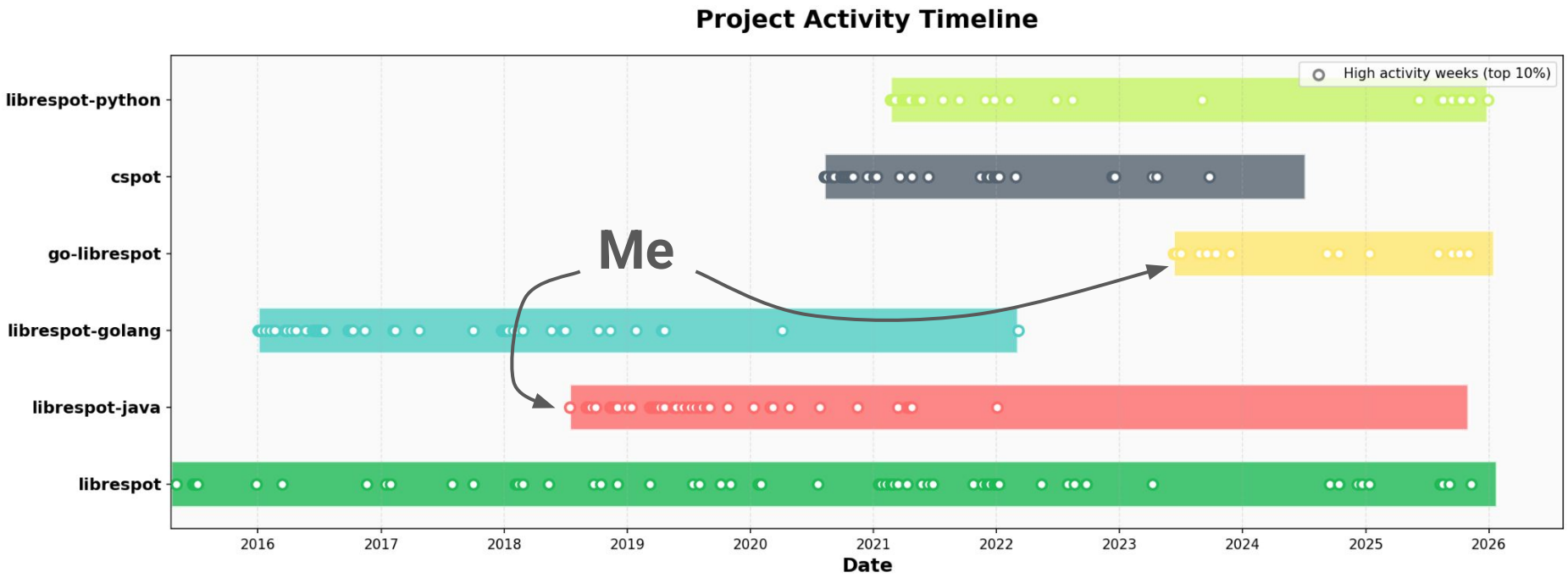
- A bulk downloader
- A way to skip or bypass ads
- A wrapper around Spotify's public APIs
- A control library
- An alternative GUI, TUI or CLI

* More on that later

Why librespot?

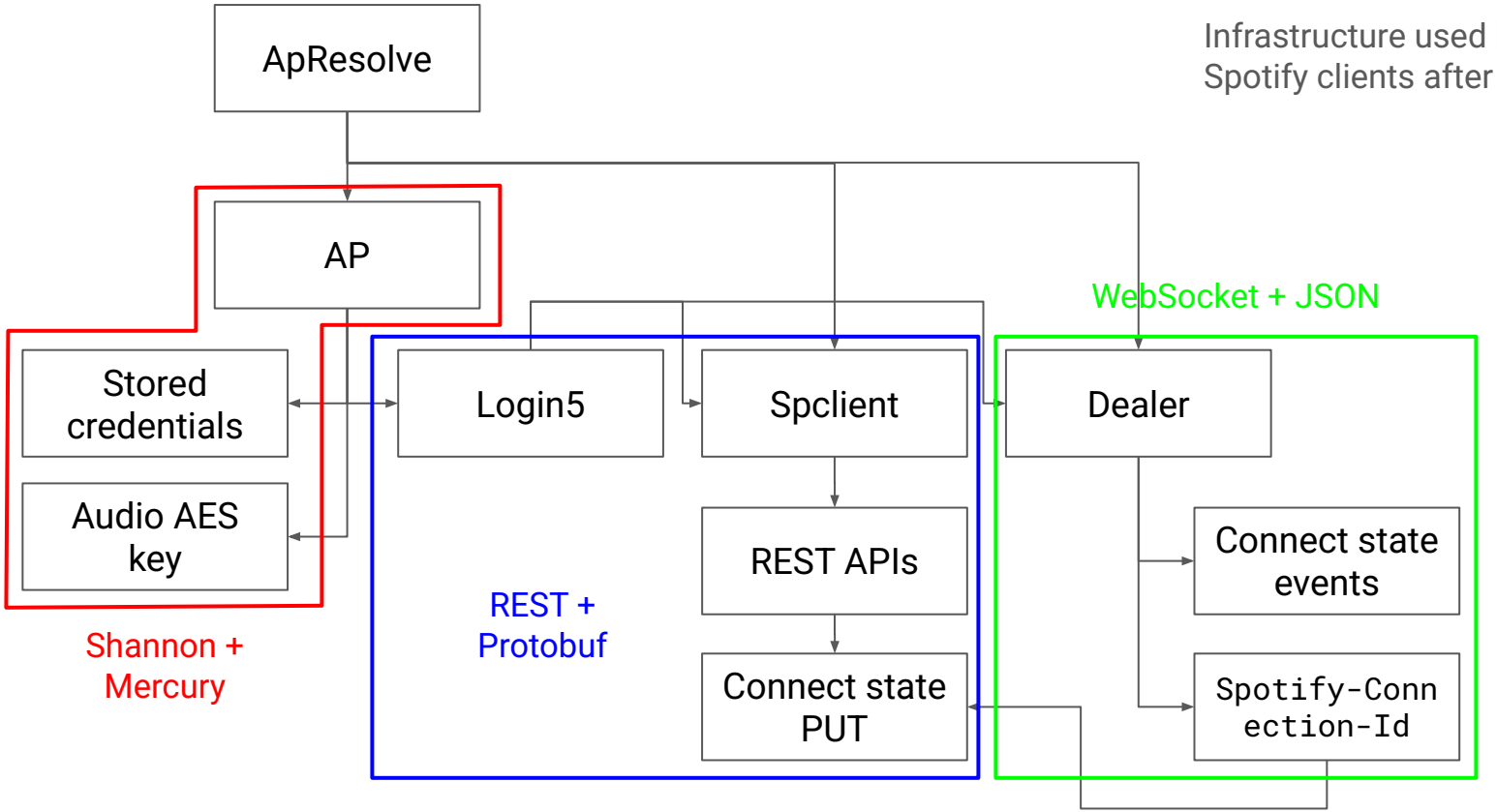
- Turn any device into a Spotify Connect device
- Headless operation, no desktop environment required
- Lightweight and low resource usage
- Open-source transparency and control, know what you are running
- Customization of audio pipelines or multi-room setups
- Home automation and integration
- DIY audio and Hi-Fi setups

The librespot family

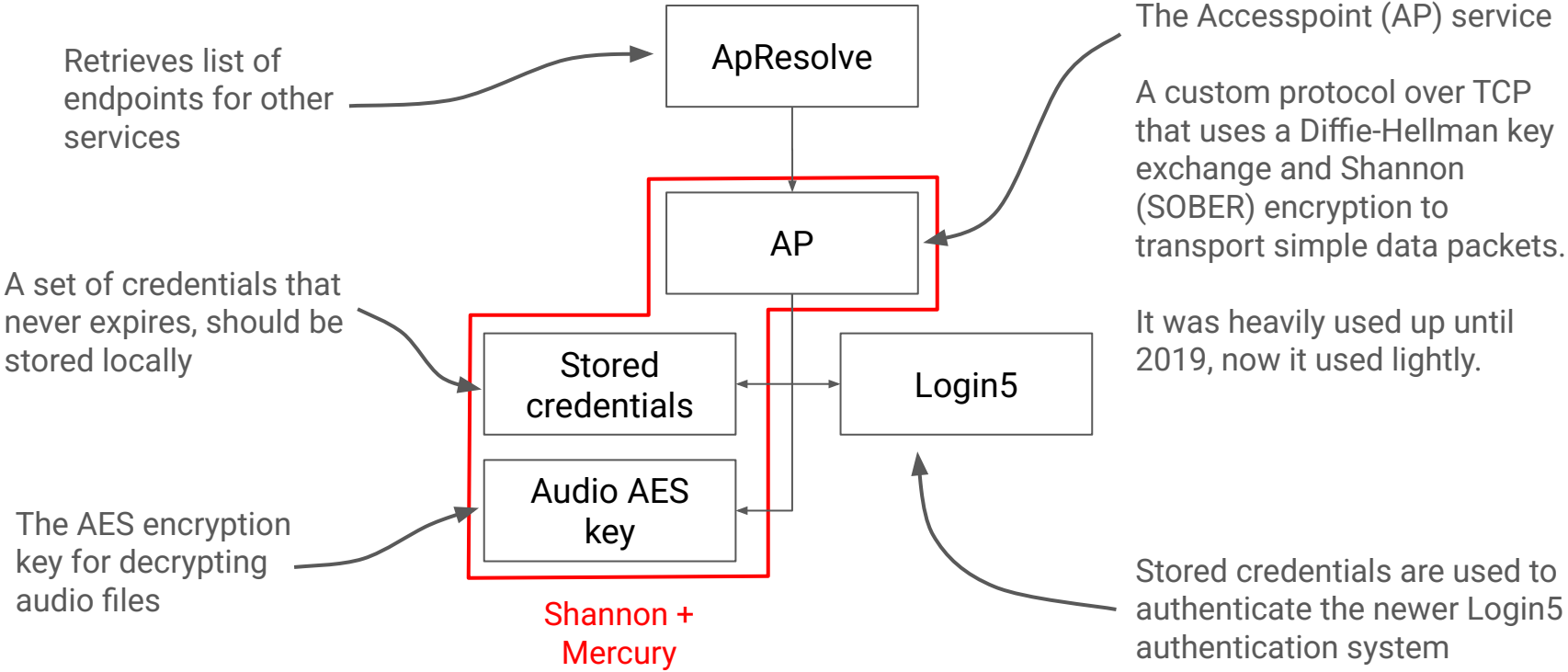


The Spotify infrastructure

Infrastructure used by Spotify clients after 2019



The Spotify infrastructure

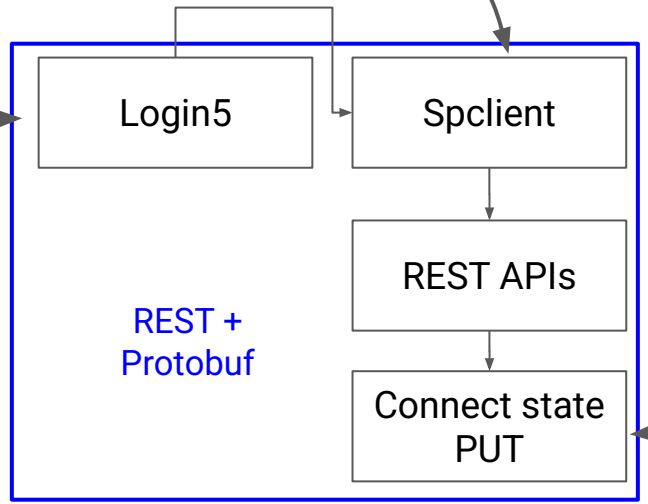


The Spotify infrastructure

Authentication system to authenticate new services

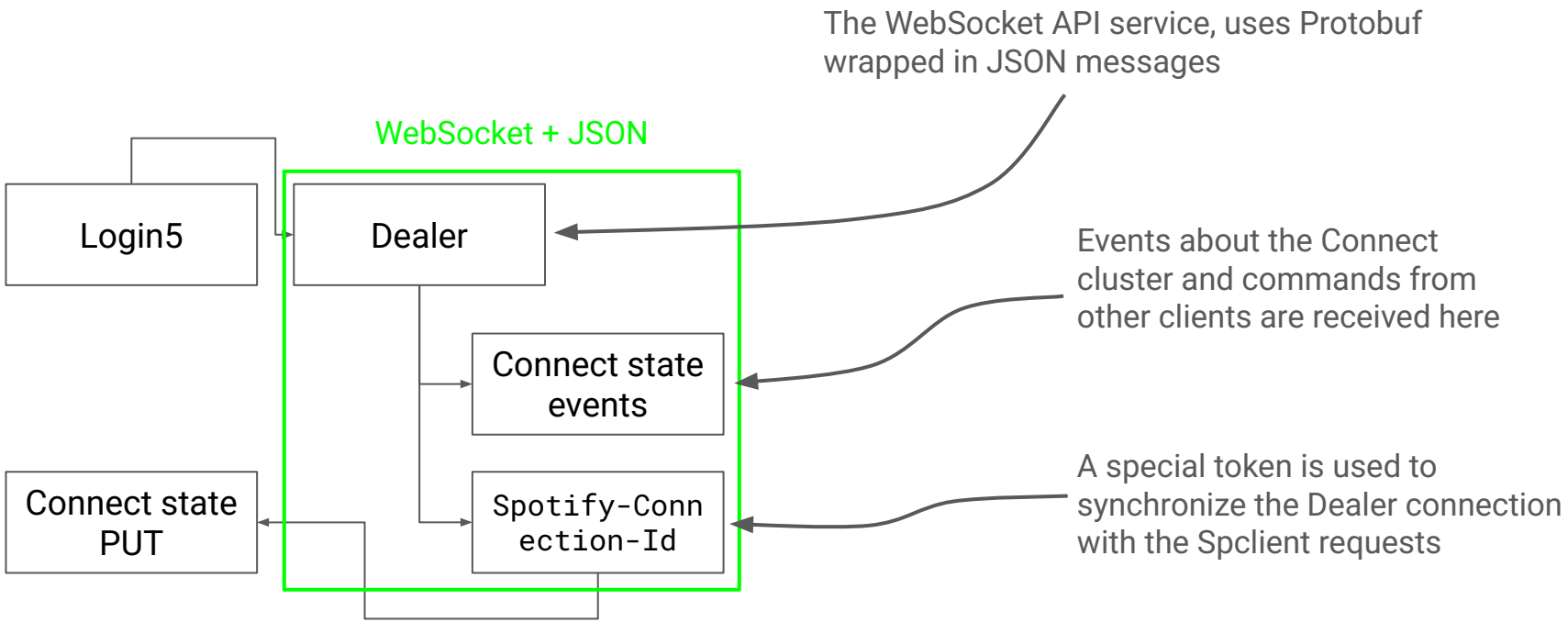
Provides Bearer tokens for everyone.

The REST API service, uses mainly Protobuf payloads



Connect state APIs publish the state of the Connect cluster to the cloud

The Spotify infrastructure



The technical challenges

- Intercepting HTTPS traffic ⬅️
- Recovering C++ Protobuf classes in Ghidra
- Logging Shannon encrypted traffic

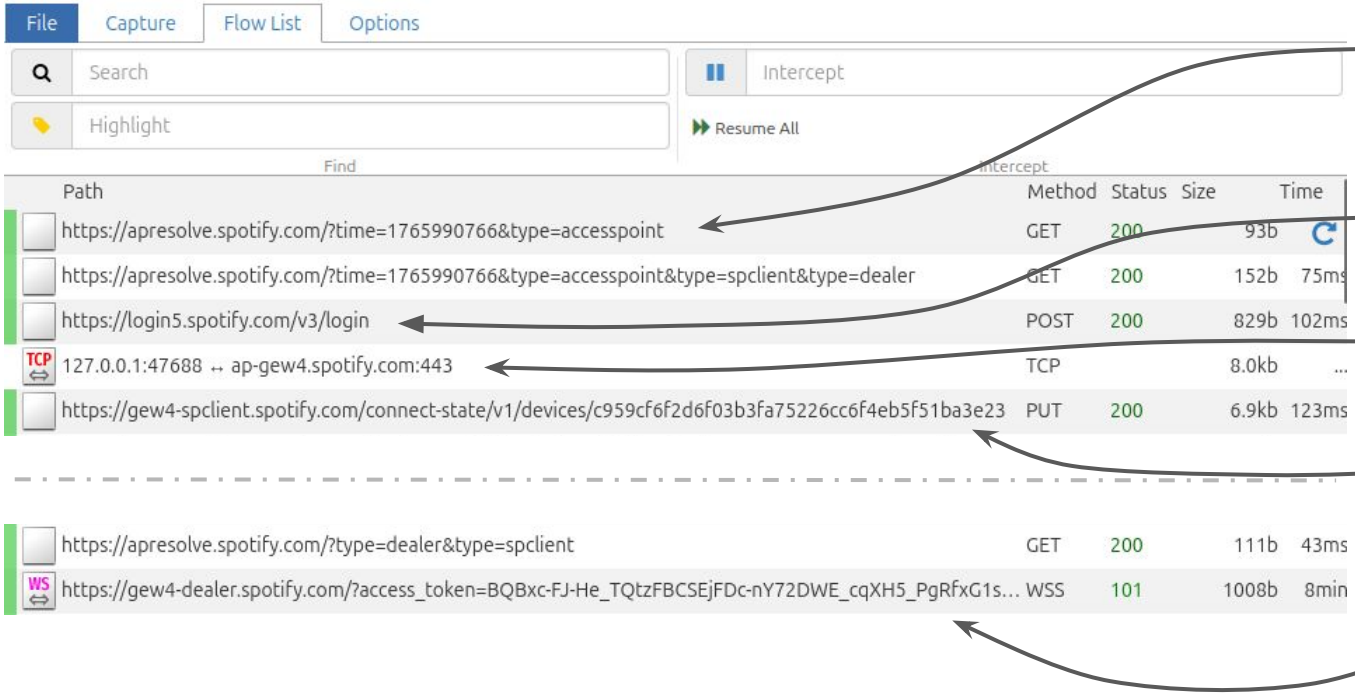
Intercepting HTTPS traffic

Nowadays Spotify uses mainly HTTPS APIs, the traffic is encrypted and cannot be passively intercepted: we need a **MITM** approach. Luckily, Spotify doesn't employ any sort of certificate pinning.

We can do the following:

- Install and start **mitmproxy/mitmproxy**
- Install the CA certificate for the system and for Chrome
- Specify the proxy URL in the Spotify Desktop client
- Voilà

Intercepting HTTPS traffic



The technical challenges

- Intercepting HTTPS traffic ✅
- Recovering C++ Protobuf classes in Ghidra ⬅️
- Logging Shannon encrypted traffic

What is Protobuf


```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

A proto definition.

```
// Java code
Person john = Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .build();
output = new FileOutputStream(args[0]);
john.writeTo(output);
```

Using a generated class to persist data.

```
// C++ code
Person john;
fstream input(argv[1],
    ios::in | ios::binary);
john.ParseFromIstream(&input);
id = john.id();
name = john.name();
email = john.email();
```

Using a generated class to parse persisted data.

Protocol Buffers are language-neutral, platform-neutral extensible mechanisms for serializing structured data.

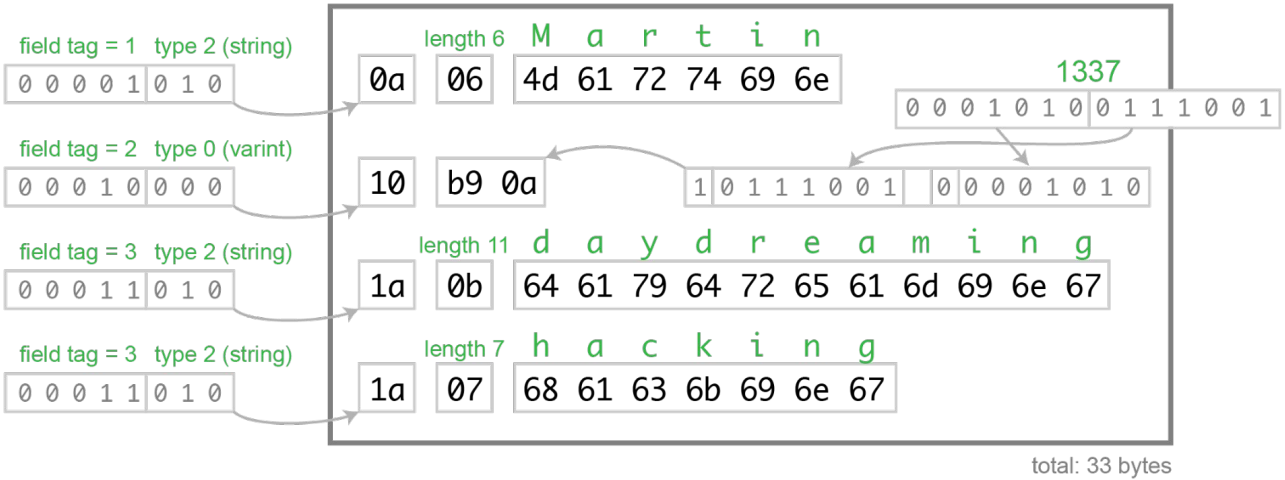
Think XML or JSON, but smaller.

Spotify has been using it for a while, at least since librespot was born.

How Protobuf works

The Protobuf wire format is **binary**. Serialized messages just use the field's number as the key. The name for each field can only be determined by referencing the **message type definition**.

```
message User {
  string name = 1;
  int32 favorite_number = 2;
  repeated string hobbies = 3;
}
```



Recovering Protobuf definitions

We can recover some Protobuf definitions by looking for the type's definition. What we are looking for is `FileDescriptorProto`, a special message that describes a `.proto` file.

When compiling Protobuf for C++, this message is embedded in the binary in its serialized form.

Using a tool like **marin-m/pbtk** against the Spotify desktop client binary extracts ~900 files and ~2400 messages.

Some of them also expose new features that haven't been released yet 👀.

Recovering Protobuf definitions


```
message FileDescriptorProto {
  optional string name = 1;
  optional string package = 2;
  repeated string dependency = 3;
  repeated DescriptorProto message_type = 4;
  optional FileOptions options = 8;
  optional string syntax = 12;
}
```

google/protobuf/any.proto

```
package google.protobuf;

option go_package = "google.golang.org/protobuf/types/known/anypb";
option java_package = "com.google.protobuf";
option java_outer_classname = "AnyProto";
option java_multiple_files = true;
option objc_class_prefix = "GPB";
option csharp_namespace = "Google.Protobuf.WellKnownTypes";

message Any {
  string type_url = 1;
  bytes value = 2;
}
```

```
{
  "syntax": "proto3",
  "name": "google/protobuf/any.proto",
  "package": "google.protobuf",
  "dependency": [],
  "messageType": [{
    "name": "Any",
    "field": [{
        "name": "type_url",
        "number": 1,
        "label": "LABEL_OPTIONAL",
        "type": "TYPE_STRING"
      }, {
        "name": "value",
        "number": 2,
        "label": "LABEL_OPTIONAL",
        "type": "TYPE_BYTES"
      }]
  }],
  "options": {
    "javaPackage": "com.google.protobuf",
    "javaOuterClassname": "AnyProto",
    "javaMultipleFiles": true,
    "goPackage": "google.golang.org/protobuf/types/known/anypb",
    "objcClassPrefix": "GPB",
    "csharpNamespace": "Google.Protobuf.WellKnownTypes"
  }
}
```

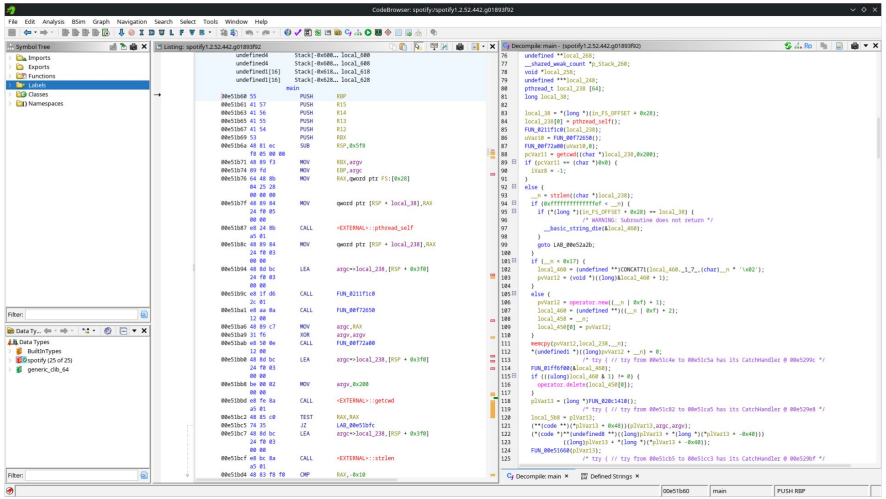
Decoded `FileDescriptorProto` for `google.protobuf.Any`

What is Ghidra

Ghidra is a software reverse engineering framework created and maintained by the NSA Research Directorate. [...] Capabilities include **disassembly**, assembly, **decompilation**, graphing, and **scripting**, along with hundreds of other features.

Decompiling the entire Spotify binary produces roughly **6 million lines** lines of C/C++ code. No meaningful function or variables names.

Where and how do we find the interesting stuff?



Recovering C++ Protobuf symbols in Ghidra

There are **multiple ways** to find interesting code paths in big binaries: from looking up interesting strings to in-depth code flow analysis. The approach that worked well for me was to identify the **usage of C++ Protobuf classes** in the decompiled code.

Each Protobuf message generates a C++ class which extends the virtual class `google::protobuf::Message`. This is good because virtual classes have **virtual tables** which are hardcoded in the binary and referenced in class constructors and destructors.

But how do we find VTABLEs, constructors and destructors and the Protobuf message they represent?

Recovering C++ Protobuf symbols in Ghidra

Our objective is to recover symbols related to Protobuf messages completely **deterministically** and, possibly, automatically.

We'll look at some generated C++ code for the `google.protobuf.Any` message.

```
const char descriptor_table_protodef_google_2fprotobuf_2fany_2eproto[] ABSL_ATTRIBUTE_SECTION_VARIABLE(
    protodesc_cold) = {
    "\n\031google/protobuf/any.proto\022\017google.prot"
    "obuf\"&\n\003Any\022\020\n\010type_url\030\001 \001(\t\022\r\n\005value\030\002"
    " \001(\014Bv\n\023com.google.protobufB\010AnyProtoP\001Z"
    ",google.golang.org/protobuf/types/known/"
    "anypb\242\002\003GPB\252\002\036Google.Protobuf.WellKnownT"
    "ypesb\006proto3"
};
```

Serialized `FileDescriptorProto` as generated C++ code for `google.protobuf.Any`

Recovering C++ Protobuf symbols in Ghidra

```
const ::_pbi::DescriptorTable descriptor_table_google_2fprotobuf_2fany_2eproto = {
    .is_initialized: false,
    .is_eager: false,
    .size: 212,
    .descriptor: descriptor_table_protodef_google_2fprotobuf_2fany_2eproto,
    .filename: "google/protobuf/any.proto",
    .once: &descriptor_table_google_2fprotobuf_2fany_2eproto_once,
    .deps: nullptr,
    .num_deps: 0,
    .num_messages: 1,
  schemas,
    .default_instances: file_default_instances,
    .offsets: TableStruct_google_2fprotobuf_2fany_2eproto::offsets,
    .file_level_metadata: file_level_metadata_google_2fprotobuf_2fany_2eproto,
    .file_level_enum_descriptors: file_level_enum_descriptors_google_2fprotobuf_2fany_2eproto,
    .file_level_service_descriptors: file_level_service_descriptors_google_2fprotobuf_2fany_2eproto,
};
```

From previous slide

Generated internal C++ descriptor table for `google.protobuf.Any`

Recovering C++ Protobuf symbols in Ghidra

```
// This function exists to be marked as weak.
// It can significantly speed up compilation by breaking up LLVM's SCC
// in the .pb.cc translation units. Large translation units see a
// reduction of more than 35% of walltime for optimized builds. Without
// the weak attribute all the messages in the file, including all the
// vtables and everything they use become part of the same SCC through
// a cycle like:
// GetMetadata -> descriptor table -> default instances ->
//   vtables -> GetMetadata
// By adding a weak function here we break the connection from the
// individual vtables back into the descriptor table.
PROTOBUF_ATTRIBUTE_WEAK const ::_pbi::DescriptorTable* descriptor_table_google_2fprotobuf_2fany_2eproto_getter() {
  return &descriptor_table_google_2fprotobuf_2fany_2eproto;
}
```

Generated C++ for `google.protobuf.Any`

From previous slide

Recovering C++ Protobuf symbols in Ghidra

From previous slide

Generated C++ for `google.protobuf.Any`

```
::google::protobuf::Metadata Any::GetMetadata() const {
  return ::_pbi::AssignDescriptors( table: &descriptor_table_google_2fprotobuf_2fany_2eproto_getter,
                                    once: &descriptor_table_google_2fprotobuf_2fany_2eproto_once,
                                    metadata: file_level_metadata_google_2fprotobuf_2fany_2eproto[0]);
}
```

Luckily this method is virtual on `google::protobuf::internal::Message`, so it'll end up in the VTABLE:

```
// Get a struct containing the metadata for the Message, which is used in turn
// to implement GetDescriptor() and GetReflection() above.
virtual Metadata GetMetadata() const = 0;
```

Recovering C++ Protobuf symbols in Ghidra

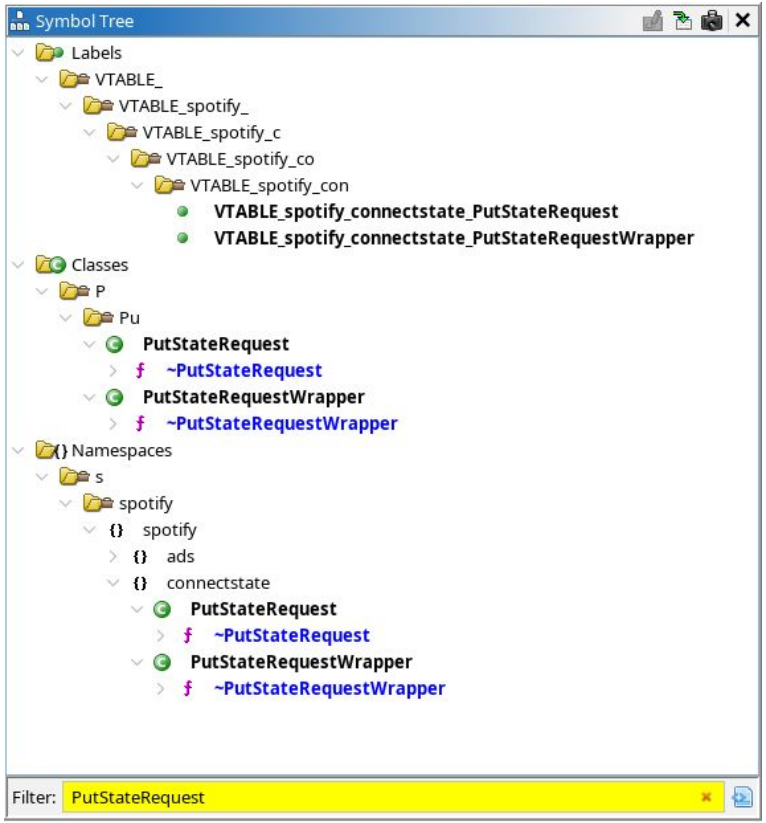
We have identified a path to go from the `FileDescriptorProto` to the VTABLE of each Protobuf message class. However, there are ~2400 Protobuf messages in the Spotify binary.

Can we automate this? **Yes**.

All the code for the Ghidra script is available at **devgianlu/GhidraProtobufCpp**. This code is quite old, but still works fairly well against Spotify.

It will not work with newer versions of the Protobuf generator as the generated C++ code has changed substantially.

Recovering C++ Protobuf symbols in Ghidra



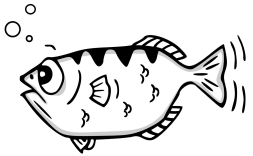
ProtoDescriptorTable_connect

02a35268	00 00 00 00 c9 1b 00 00 00 41 84 00 ...	Descripto...		
02a35268	00	bool	FALSE	is_initialized
02a35269	00	bool	FALSE	is_eager
02a3526a	00	undefined1	00h	field2_0x2
02a3526b	00	undefined1	00h	field3_0x3
02a3526c	c9 1b 00 00	int	1BC9h	size
02a35270	00 41 84 00 00 00 00 00	char *	DAT_00844100	descriptor
02a35278	f1 29 50 00 00 00 00 00	char *	s_connect.proto_005029f1	filename
02a35280	a4 ff 06 03 00 00 00 00	void *	DAT_0306ffa4	once
02a35288	50 ff 9c 02 00 00 00 00	Descripto...	PTR_ProtoDescriptorTable_devices_...	deps
02a35290	03 00 00 00	int	3h	num_deps
02a35294	13 00 00 00	int	13h	num_messages
02a35298	d0 5c 84 00 00 00 00 00	void *	DAT_00845cd0	schemas
02a352a0	70 ff 9c 02 00 00 00 00	void * *	PTR_PTR_029cff70	default_instances
02a352a8	50 3b 84 00 00 00 00 00	uint *	DAT_00843b50	offsets
02a352b0	50 fe 06 03 00 00 00 00	void *	DAT_0306fe50	file_level_metadata
02a352b8	80 ff 06 03 00 00 00 00	void * *	DAT_0306ff80	file_level_enum_descriptors
02a352c0	00 00 00 00 00 00 00 00	void * *	00000000	file_level_service_descriptors

The technical challenges

- Intercepting HTTPS traffic ✅
- Recovering C++ Protobuf classes in Ghidra ✅
- Logging Shannon encrypted traffic ⬅️

What is GDB



We'll use the **GNU Debugger** to go deeper than what we could with static analysis. This way we can make or confirm assumptions derived from analysing the decompiled code by checking what data is actually being processed by the code.

Using GDB without debug symbols is a completely different experience, we can use tools like **pwndbg/pwndbg** to facilitate reverse engineering.

```
(gdb) start
Temporary breakpoint 1 at 0x1151
Starting program: /home/devgianlu/Downloads/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, 0x0000555555555151 in main ()
```

```
pwndbg> start
Temporary breakpoint 1 at 0x1151
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, 0x0000555555555151 in main ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
──────────────────────────────────────────────────────────────────────────────[ LAST SIGNAL ]──
Breakpoint hit at 0x555555555151
──────────────────────────────────────────────────────────────[ REGISTERS / show-flags off / show-compact-regs off ]──
 RAX  0x555555555149 (main) ◂— endbr64
 RBX  0x7fffffffe698 —▸ 0x7fffffffe99e ◂— '/home/devgianlu/Downloads/a.out'
 RCX  0x5555555557dc0 (__do_global_dtors_aux_fini_array_entry) —▸ 0x5555555555100 (__do_global_dtors_aux) ◂— endbr64
 RDX  0x7fffffffe6a8 —▸ 0x7fffffffe9be ◂— 'XDG_DATA_DIRS=/usr/local/share/:/usr/share/:/var/lib/snapd/desktop'
 RDI  1
 RSI  0x7fffffffe698 —▸ 0x7fffffffe99e ◂— '/home/devgianlu/Downloads/a.out'
 R8   0
 R9   0x7ffff7fca380 (_dl_fini) ◂— endbr64
 R10  0x7ffffffe290 ◂— 0x800000
 R11  0x203
 R12  1
 R13  0
 R14  0x5555555557dc0 (__do_global_dtors_aux_fini_array_entry) —▸ 0x5555555555100 (__do_global_dtors_aux) ◂— endbr64
 R15  0x7ffff7ffd000 (_rtld_global) —▸ 0x7ffff7ffe2e0 —▸ 0x555555554000 ◂— 0x10102464c457f
 RBP  0x7fffffffe570 —▸ 0x7fffffffe610 —▸ 0x7fffffffe670 ◂— 0
 RSP  0x7fffffffe570 —▸ 0x7fffffffe610 —▸ 0x7fffffffe670 ◂— 0
 RIP  0x555555555151 (main+8) ◂— lea rax, [rip + 0xeac]
──────────────────────────────────────────────────────────────────[ DISASM / x86-64 / set emulate on ]──
► 0x555555555151 <main+8>     lea    rax, [rip + 0xeac]     RAX => 0x555555556004 ◂— 'Hello World!'
  0x555555555158 <main+15>    mov    rdi, rax               RDI => 0x555555556004 ◂— 'Hello World!'
  0x55555555515b <main+18>    mov    eax, 0                 EAX => 0
  0x555555555160 <main+23>    call   printf@plt                    <printf@plt>

  0x555555555165 <main+28>    nop
  0x555555555166 <main+29>    pop    rbp
  0x555555555167 <main+30>    ret

  0x555555555168 <_fini>      endbr64
  0x55555555516c <_fini+4>    sub    rsp, 8
  0x555555555170 <_fini+8>    add    rsp, 8
  0x555555555174 <_fini+12>   ret
──────────────────────────────────────────────────────────────────────────────[ STACK ]──
00:0000│ rbp rsp 0x7fffffffe570 —▸ 0x7fffffffe610 —▸ 0x7fffffffe670 ◂— 0
01:0008│ +008    0x7fffffffe578 —▸ 0x7ffff7c2a1ca (__libc_start_call_main+122) ◂— mov edi, eax
02:0010│ +010    0x7fffffffe580 —▸ 0x7fffffffe5c0 —▸ 0x5555555557dc0 (__do_global_dtors_aux_fini_array_entry) —▸ 0x5555555555100 (__do_global_dtors_aux) ◂— endbr64
03:0018│ +018    0x7fffffffe588 —▸ 0x7fffffffe698 —▸ 0x7fffffffe99e ◂— '/home/devgianlu/Downloads/a.out'
04:0020│ +020    0x7fffffffe590 ◂— 0x155554040
05:0028│ +028    0x7fffffffe598 —▸ 0x555555555149 (main) ◂— endbr64
06:0030│ +030    0x7fffffffe5a0 —▸ 0x7fffffffe698 —▸ 0x7fffffffe99e ◂— '/home/devgianlu/Downloads/a.out'
07:0038│ +038    0x7fffffffe5a8 ◂— 0x3433ba88695e8466
──────────────────────────────────────────────────────────────────────────[ BACKTRACE ]──
► 0   0x555555555151 main+8
  1   0x7ffff7c2a1ca __libc_start_call_main+122
  2   0x7ffff7c2a28b __libc_start_main+139
  3   0x5555555550b5 _start+37
```

What is Frida

It's a dynamic code **instrumentation** toolkit: it lets you inject snippets of JavaScript into native apps on Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD, and QNX.

Typically, you'll use GDB to manually inspect the program and look around interactively while the program is running. Then, with Frida you can write some **JavaScript** code that does those things automatically for you.

We'll have a look at the Ghidra -> GDB -> Frida workflow to intercept Shannon traffic before it's encrypted and sent and after it's been received and decrypted.

What is the Shannon cipher

It is part of the SOBER family of stream ciphers initially designed by QUALCOMM Australia starting in 1997. The reference implementation is still available on QUALCOMM website through Wayback Machine.

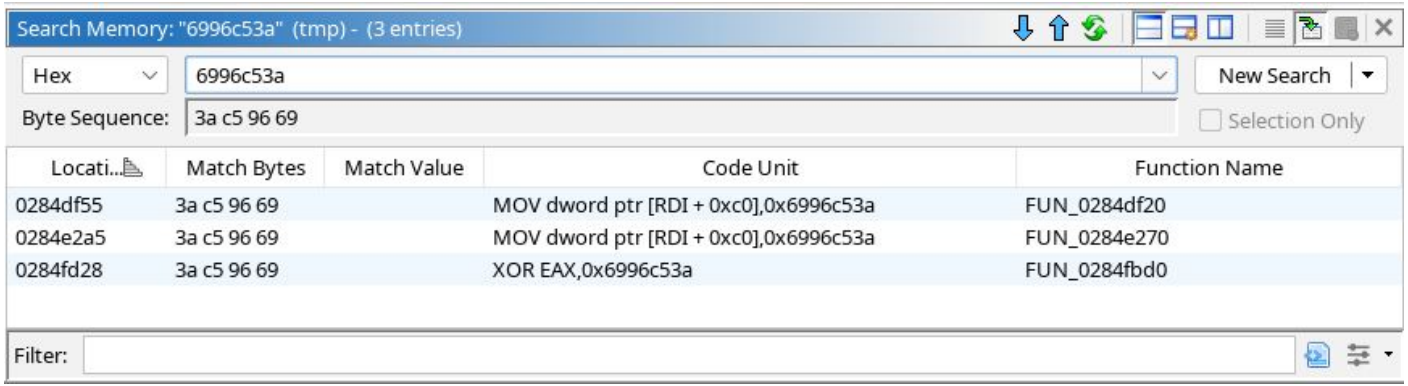
The cipher performs encryption and message authentication simultaneously.

The reference implementation is quite simple to use and has just a few methods:

```
void shn_key(shn_ctx *c, const uint8_t key[], int keylen); /* set key */
void shn_nonce(shn_ctx *c, const uint8_t nonce[], int nlen); /* set Init Vector */
void shn_encrypt(shn_ctx *c, uint8_t *buf, int nbytes); /* encrypt + MAC */
void shn_decrypt(shn_ctx *c, uint8_t *buf, int nbytes); /* decrypt + MAC */
void shn_finish(shn_ctx *c, uint8_t *buf, int nbytes); /* finalise MAC */
```

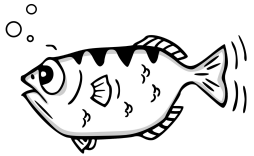
Finding Shannon code in Ghidra

We can search for the special constant `0x6996c53a` used by the Shannon cipher and perform some additional static analysis to figure out where those functions reside in the binary.



Now we just need to verify that we found them in GDB.

Verifying with GDB



Let's verify our assumptions with some **dynamic analysis**.

We'll attach GDB to Spotify, set a breakpoint at the address of the `shn_encrypt` function and check its arguments.

```
void shn_encrypt(shn_ctx *c, uint8_t *buf, int nbytes);
```

```
Thread 60 "Network Thread" hit Breakpoint 2, 0x0000555557ca22d0 in ?? ()
pwndbg> p/x $rsi
$1 = 0x7fff40009710
pwndbg> vmmap 0x7fff40009710
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
             Start                End Perm     Size  Offset File (set vmmap-prefer-relpaths on)
      0x7fff3c021000     0x7fff40000000 ---p  3fdf000       0 [anon_7fff3c021]
►     0x7fff40000000     0x7fff40021000 rw-p    21000       0 [anon_7fff40000] +0x9710
      0x7fff40021000     0x7fff44000000 ---p  3fdf000       0 [anon_7fff40021]
pwndbg> p $edx
$2 = 398
pwndbg> p/x (char[398])*0x7fff40009710
$3 = {0xab, 0x1, 0x8b, 0x52, 0xc0, 0x1, 0x52, 0xb, 0x31, 0x31, 0x31, 0x34, 0x35, 0x30, 0x38, 0x39, 0x30, 0x31, 0x39, 0xa0, 0x1, 0x1, 0xf2, 0x1, 0xac, 0x1, 0x41, 0x67, 0x42, 0x6f,
  0x35, 0x36, 0x78, 0x54, 0x57, 0x69, 0x6c, 0x6b, 0x75, 0x6b, 0x64, 0x6a, 0x4d, 0x67, 0x36, 0x4b, 0x7a, 0x38, 0x4b, 0x48, 0x79, 0x47, 0x56, 0x44, 0x49, 0x72, 0x73, 0x78, 0x43, 0x45,
  0x2d, 0x31, 0x57, 0x4b, 0x49, 0x46, 0x61, 0x47, 0x78, 0x68, 0x5a, 0x6b, 0x48, 0x41, 0x43, 0x6b, 0x58, 0x51, 0x32, 0x56, 0x37, 0x73, 0x71, 0x52, 0x70, 0x5a, 0x30, 0x36, 0x77, 0x5f,
  0x70, 0x6a, 0x68, 0x61, 0x78, 0x74, 0x64, 0x75, 0x4a, 0x6e, 0x63, 0x43, 0x7a, 0x6f, 0x72, 0x77, 0x69, 0x70, 0x73, 0x32, 0x66, 0x6b, 0x45, 0x6a, 0x54, 0x30, 0x47, 0x36, 0x4c, 0x48,
  0x6d, 0x6f, 0x63, 0x41, 0x34, 0x49, 0x57, 0x30, 0x2d, 0x5a, 0x4c, 0x37, 0x76, 0x68, 0x4c, 0x73, 0x56, 0x77, 0x30, 0x6d, 0x6a, 0x46, 0x59, 0x49, 0x67, 0x58, 0x75, 0x69, 0x30, 0x72,
  0x66, 0x78, 0x62, 0x59, 0x6a, 0x58, 0x39, 0x65, 0x36, 0x7a, 0x33, 0x58, 0x7a, 0x73, 0x55, 0x7a, 0x6a, 0x37, 0x63, 0x54, 0x76, 0x63, 0x58, 0x4a, 0x6e, 0x62, 0x2d, 0x6c, 0x69, 0x6b,
  0x44, 0x7a, 0x69, 0x33, 0x66, 0x66, 0x38, 0x77, 0x74, 0x48, 0x74, 0x7a, 0x75, 0x42, 0x45, 0x4b, 0x4c, 0x47, 0x5a, 0x0...}
```

Setting up Frida

My setup uses a VM to run the Spotify binary, but that is no problem for Frida which can run a server that exposes a TCP port and is able to **spawn executables while instrumenting them**.

For example, we can launch the Spotify binary and get its base address from the REPL.

This is not really practical, **we can do better**.

```
devgianlu@home-fedora ~> frida -H 192.168.100.140 --kill-on-exit /usr/bin/spotify
     ____
    / _  |   Frida 16.5.9 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to 192.168.100.140 (id=socket@192.168.100.140)
Spawned `/usr/bin/spotify`. Resuming main thread!
[Remote::spotify ]-> Module.getBaseAddress("spotify")
"0x5ff5fc676000"
[Remote::spotify ]->
```

The Frida script


```
const baseAddr :NativePointer  = Module.getBaseAddress( name: "spotify")
console.log(`base addr = ${baseAddr}`)
```

```
Interceptor.attach(baseAddr.add(0x274e2d0),  callbacksOrProbe: function () :void  {
  const ctx = this.context
  const len :number  = ctx.rdx.toUInt32()
  const data :Uint8Array<ArrayBuffer>  = readUint8Array(ctx.rsi, len)
  console.log(`shn_encrypt(${ellipse(u8hex(data))}, ${len})`);
})
```

```
Interceptor.attach(baseAddr.add(0x0274ef70),  callbacksOrProbe: {
  onEnter: function () :void  {
    this.originalRsi = this.context.rsi
    this.originalRdx = this.context.rdx
  },
  onLeave: function () :void  {
    const len :number  = this.originalRdx.toUInt32()
    const data :Uint8Array<ArrayBuffer>  = readUint8Array(this.originalRsi, len)
    if (len > 0) console.log(`shn_decrypt(${ellipse(u8hex(data))}, ${len})`);
  }
})
```

We can use the `Interceptor` module to hook into the `shn_encrypt` and `shn_decrypt` functions and read the program registers and memory.

For decryption we are interested in the state at the end of the function, after decryption has been performed, thus we save the registers and use `onLeave`.

The Frida script


```
Spawning `/usr/bin/spotify`...
base addr = 0x5f6f0b873000
Spawned `/usr/bin/spotify`. Resuming main thread!
[Remote::spotify ]-> shn_encrypt(ab018b52c001520b3131313435303839303139a00101f201ac014167426f3536..., 398)
shn_decrypt(ac00f3, 3)
shn_decrypt(520b3131313435303839303139a00101c80100f00101c202ac014167426f3536..., 243)
shn_encrypt(0f00140000000000000000000000000000000000000000, 23)
shn_decrypt(040004, 3)
shn_decrypt(69692699, 4)
shn_decrypt(020150, 3)
shn_decrypt(696926996969971900cadf04abd2d928000000000000000000000000000000..., 336)
shn_decrypt(760002, 3)
shn_decrypt(0000, 2)
shn_decrypt(1b0002, 3)
shn_decrypt(4954, 2)
shn_decrypt(5012a9, 3)
shn_decrypt(3c3f786d6c2076657273696f6e3d27312e302720656e636f64696e673d277574..., 4777)
shn_decrypt(1f0011, 3)
shn_decrypt(00000000000000000000000000000000, 17)
shn_decrypt(690000, 3)
shn_encrypt(b200e900080000000000000001010002005f0a1c686d3a2f2f6576656e742d73..., 236)
shn_decrypt(b501c7, 3)
shn_decrypt(000803df4e00000000001000101b80ad901686d3a2f2f7075736865722f7631..., 455)
shn_decrypt(100014, 3)
shn_decrypt(00000000c81d59e8284303caeacadf8491a6faba, 20)
shn_decrypt(b20030, 3)
shn_decrypt(000800000000000000010100010021 0a1c686d3a2f2f6576656e742d73657276..., 48)
[Remote::spotify ]->
```

-> Login

<- APWelcome

<- CountryCode

<- ProductInfo

-> MercuryReq

<- MercuryEvent

<- MercuryReq

The technical challenges

- Intercepting HTTPS traffic ✅
- Recovering C++ Protobuf classes in Ghidra ✅
- Logging Shannon encrypted traffic ✅

The legal challenges

The projects are constantly at risk of getting deleted if we do something that makes Spotify angry. They could find a way to wipe everything because **they have the legal strength** and we don’t.

For this reason, some features will never be available publicly across any of librespot projects:

- Listen reporting
- Lossless playback
- Ads playback (free accounts)

Listen reporting

Tracks played by any of the librespot projects are not accounted for and do not appear in the listening history. This is bad because **artists will not be credited** for listening to their tracks.

Spotify has a different set of APIs for reporting whether a track has been played and for how long; updating the Connect state or fetching the tracks does nothing.

Doing any work towards the reverse-engineering and implementation of listen reporting would make the project at risk because of the implications of having such code available publicly. For example, people trying to create **listen bots** to boost artists for money.

Lossless playback

Spotify recently launched lossless playback giving users access to **FLAC files**, finally joining the game of HiFi streaming providers.

This is an heavily requested feature, especially in the DIY field, by those that are focused on building hardware to get the best audio quality, or by those that want to get the most out of their already quite expensive subscription.

However, Spotify doesn't want us to mess with it* as it's protected by their **new DRM**, that we'll call StopStop.

For context, other HiFi streaming providers don't use DRMs for their audio files.

* More on that later

The StopStop DRM

As mentioned in the beginning, the audio files are encrypted with AES-CTR where the key is provided by some backend service. Originally, the decryption key was served as-is and without much constraints.

More recently they have started **cracking down on the usage of this old API** to prevent misuses. In doing so they started killing some of their own partner products.

The StopStop DRM adds a new endpoint serving an **obfuscated decryption key**. The de-obfuscation code contains some constants and procedures that they can claim for **intellectual property infringement**.

Ads playback

All the librespot projects do not support free accounts. This choice makes us more free in the decisions that we make and less worried about Spotify:

- Relieves the pressure from Spotify since we are **not stealing any revenue**
- Doesn’t force us to reverse engineer code that Spotify tries to hide, wasting time that could be spent on new features
- We don’t need to implement additional and frequently changing logic for ads playback
- Keeps away contribution and interest from modders that will most likely bring problems

How I got *temporarily* banned from Spotify

- 29th October: Discord server is created to work on StopStop
- 3rd November: First working implementation of StopStop
- 4th November: I add FLAC and StopStop support to go-librespot, no public de-obfuscation code. I start using it privately to stream FLAC files
- 5th November: Spotify sends an an email to the maintainers of the librespot projects intimating to stop working on StopStop
- 15th November: My account gets banned, I appeal to my suspension
- 17th November: I get my account back, have not used StopStop since

How you can help

There are multiple things you can do to support the projects:

- Give them a try, use them, let Spotify know
- Contribute with bug reports and feature requests
- Write some code for a bug or a new feature

If you work at Spotify:

- Get in touch
- Don’t ban me again

Thank you!
Questions?

@devgianlu - FOSDEM 2026