Binary Analysis

Introduction to Frida and Angr

Start Week 13 →


```
f4a1 8bd0 23cf 79e0 1174 c39a 26d7 b901
9f02 651a 88d3 1ab5 ef4c a918 6b7c 3ffe
3a1e e823 4ff9 02d8 becb 5f90 a4d2 67aa
f4a1 8bd0 23cf 79e0 1174 c39a 26d7 b901
```

Binary Instrumentation

For the first lab we'll be talking about **Binary Instrumentation**, the process of inserting behavior into compiled programs to observe or modify their behavior.

Increasingly popular set of techniques

- Monitoring program performance
- Tracing program execution
- Tracking memory allocations for leaks and frees
- Tracking memory accesses for indicators of vulnerabilities
- Disabling unwanted features like license checks

Binary Instrumentation

The main challenge is the technical complexity of inserting your desired functionality in to the target program.

There's a few approaches

- Alter source or IR during compilation
- Alter shared library calls
- Alter the program on disk
- Alter the program while it's running

Instrumentation: Source Transformations

Inserting hooks into either the original source or an intermediate representation during compilation.

aflplus.plus/docs/fuzzing_in_depth/

Fuzzing in Depth

Fuzzing with AFL++

The following describes how to fuzz with a target if source code is available. If you have a binary-only target, go to /

Fuzzing source code is a three-step process:

1. Compile the target with a special compiler that prepares the target to be fuzzed efficiently. This step is called "i
2. Prepare the fuzzing by selecting and optimizing the input corpus for the target.
3. Perform the fuzzing of the target by randomly mutating input and assessing if that input was processed on a n

0. Common sense risks

Please keep in mind that, similarly to many other computationally-intensive tasks, fuzzing may put a strain on your

Instrumentation: Shared Libraries

Insert hook points at shared library calls: DLL hijacking, DLL injection, `LD_PRELOAD`, or `ptrace`.

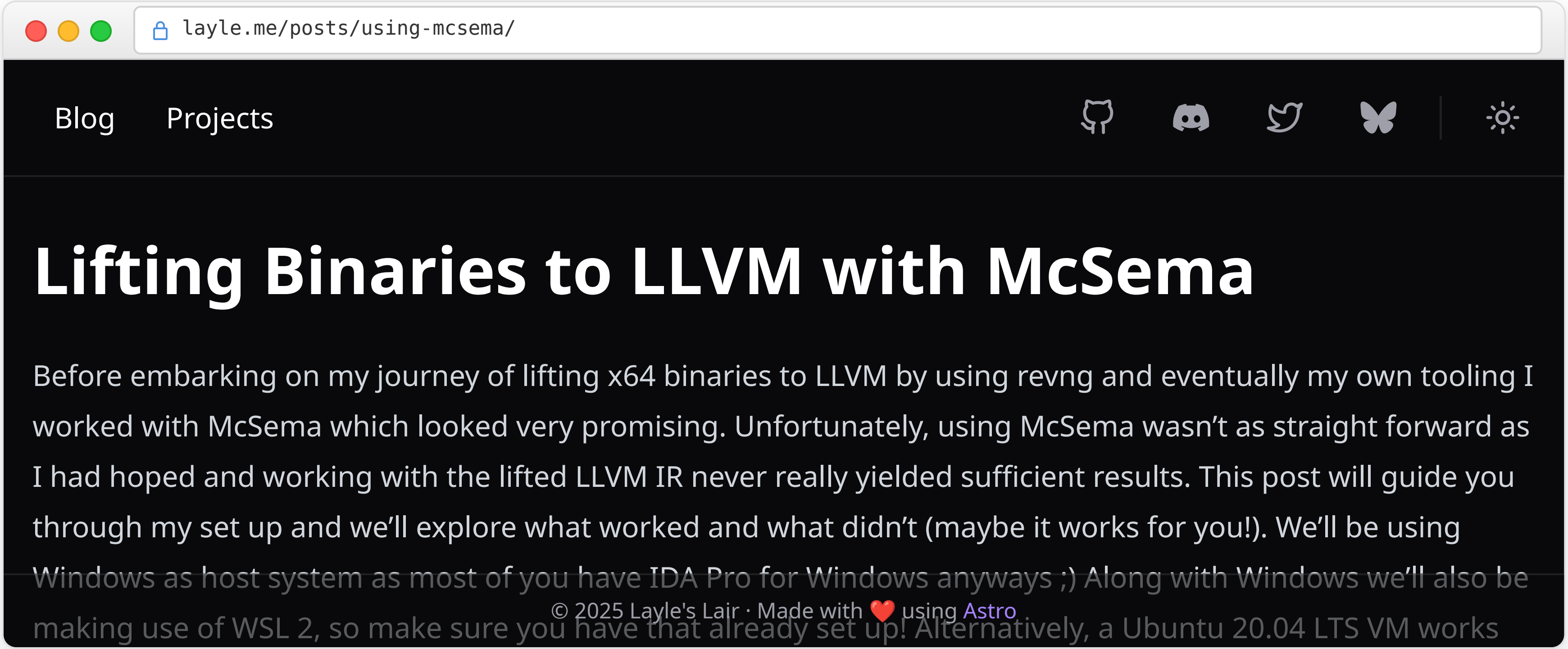
https://www.kernel.org/doc/ols/2007/ols2007v1-pages-41-52.pdf

linux.die.net/man/1/ltrace

Instrumentation: Binary Rewriting

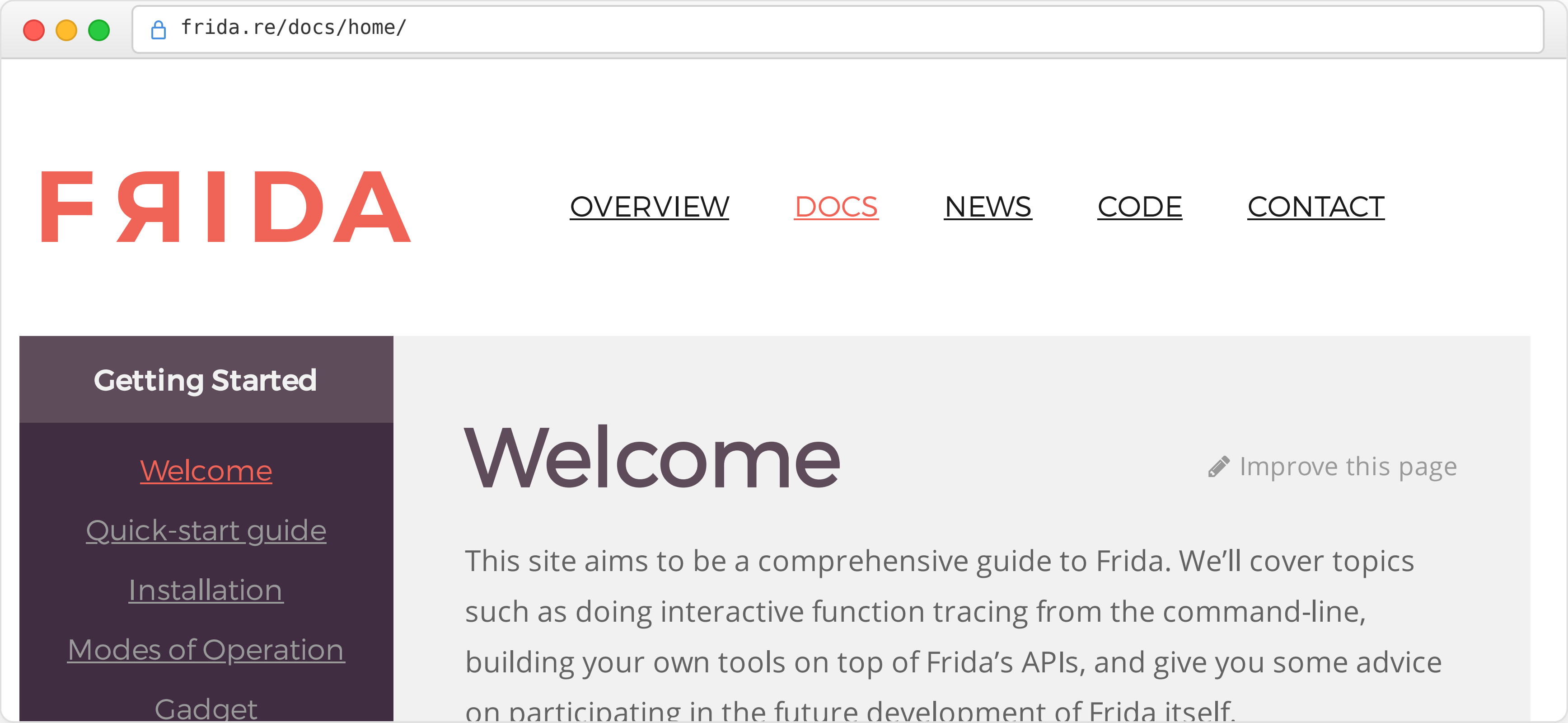
Altering the program on-disk prior to execution.

https://recon.cx/2014/slides/McSema.pdf



Instrumentation: Altering Running Programs

Hooking and modifying a program while it's running.



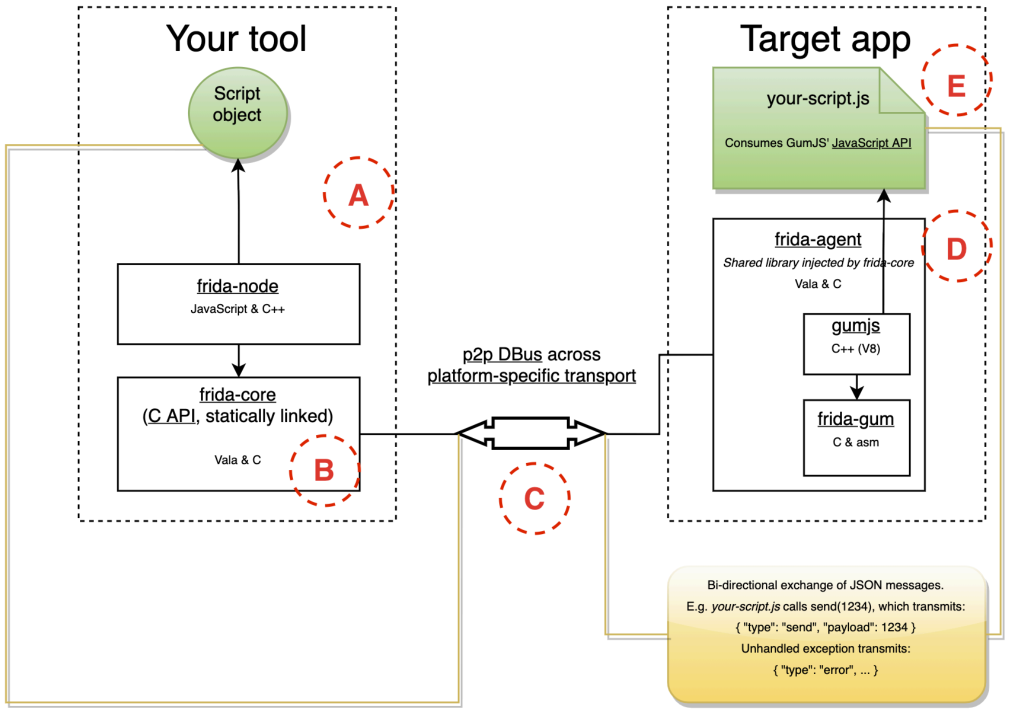
Frida Architecture

Dynamic instrumentation toolkit for developers, reverse engineers, and security researchers. Works by injecting a target process and modifying it while it's running.

- `frida-core`: Orchestrate using a command line interface or python script
  - Supports device enumeration, process enumeration, process injection, etc.
- `frida-agent`: Injects a shared object into the target process to instrument at runtime
  - Includes a javascript interpreter so it's scriptable
  - Supports tracing, function hooking, module enumeration, etc.

Frida Architecture

The core library connects to the agent library over a bus.



Frida Snippets

Script to attach frida and load a script that scans the target's memory.

```
device = frida.get_local_device()
session = device.attach(pid)
script = session.create_script(SCRIPT_PATH.read_text())
script.load()

time.sleep(5)
session.detach()
```

Script to enumerate modules.

```
Process.enumerateModulesSync().filter(m => m.path.startsWith('/data')).forEach(m => {
    var pattern = str.split('').map(letter => letter.charCodeAt(0).toString(16)).join(' ');
    try {
      var res = Memory.scanSync(m.base, m.size, pattern);
      if (res.length > 0)
        console.log(JSON.stringify({m, res}));
    } catch (e) {
      console.warn(e);
    }
  });
```

Frida Snippets

Script to enumerate ranges.

```
Process.enumerateRanges('rw-', {
  onMatch: function (range) {
    var fname = `/sdcard/${range.base}_dump`;
    var f = new File(fname, 'wb');
    f.write(instance.base.readByteArray(instance.size));
    f.flush();
    f.close();
    console.log(`base=${range.base} size=${range.size} prot=${range.protection} fname=${fname}`);
  },
  onComplete: function () {}
});
```

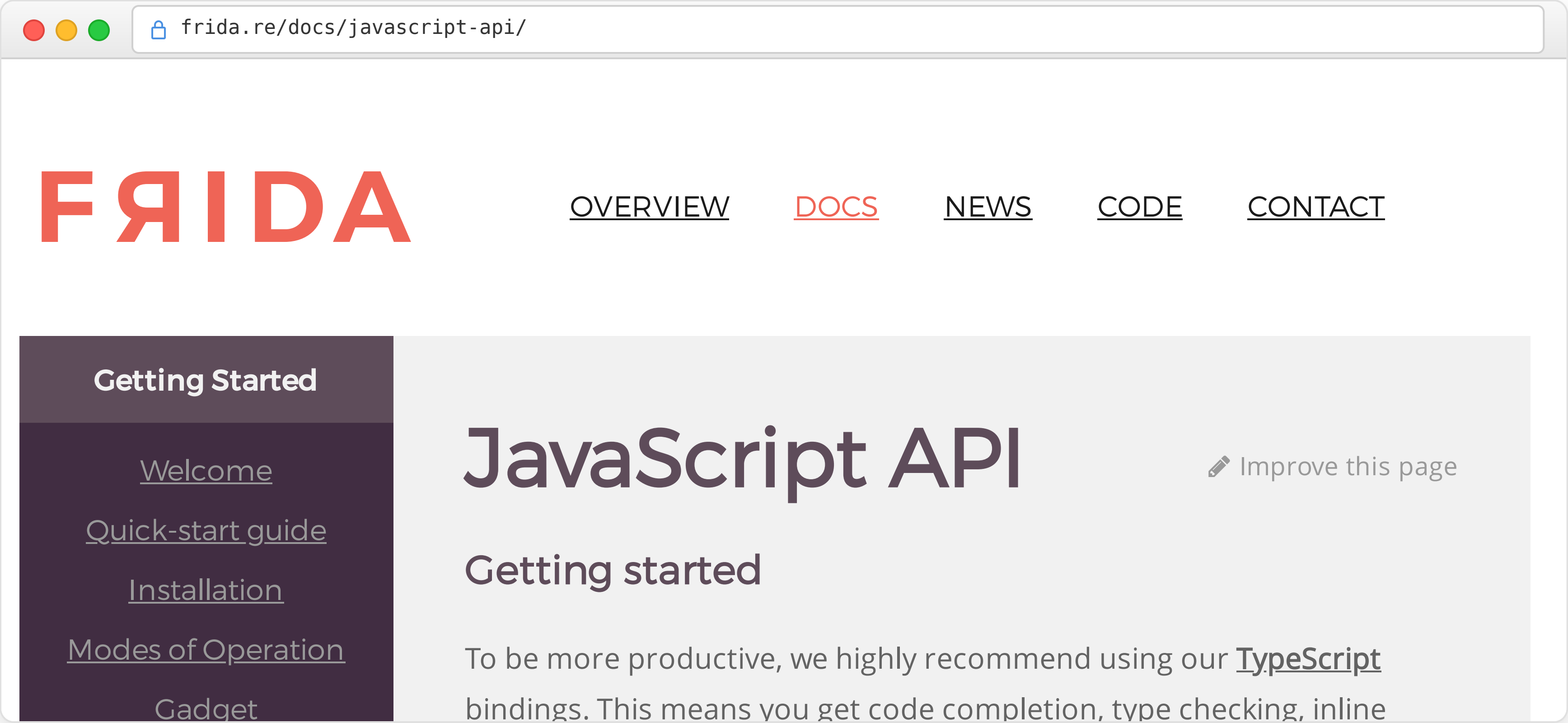
Frida Snippets

Script to monitor socket activity.

```
Process
  .getModuleByName({ linux: 'libc.so', darwin: 'libSystem.B.dylib', windows: 'ws2_32.dll' }[Process.platform])
  .enumerateExports()
  .filter(ex => ex.type === 'function' && ['connect', 'recv', 'send', 'read', 'write']
      .some(prefix => ex.name.indexOf(prefix) === 0))
  .forEach(ex => {
    Interceptor.attach(ex.address, {
      onEnter: function (args) {
        var fd = args[0].toInt32();
        var socktype = Socket.type(fd);
        if (socktype !== 'tcp' && socktype !== 'tcp6')
          return;
        var address = Socket.peerAddress(fd);
        if (address === null)
          return;
        console.log(fd, ex.name, address.ip + ':' + address.port);
      }
    })
  })
```

Frida Documentation

Use the API documentation to write you own scripts.



Lab 1

https://hacs408e.net/labs/week-13/lab-1/

1137

MINUTES

− +

Ends at 6:40 PM

Symbolic execution

Program analysis technique where program inputs are treated as symbols rather than concrete values, to explore many possible execution paths at once and derive logical constraints describing when each path is taken.

- Automatic test case generation
- Finding complex inputs to extend fuzzing coverage
- Verifying program code can or can’t reach a particular state
- Detecting potentially vulnerable memory access patterns
- Automatic exploit generation

Symbolic Execution

How symbolic executors work at a high level.

- Emulate a program on symbolic inputs (like variables x, y) instead of specific numbers (like 5, 42), so each input stands for many possible values
- As the program branches (if, while, etc.), symbolic execution collects path conditions—logical formulas that describe what must be true for that path to be followed (e.g., x > 0 && y == 3)
- A constraint solver (e.g., an SMT solver) is used to find concrete input values that satisfy these path conditions, turning abstract paths into real test cases

Its main challenges include the path explosion problem (too many paths), dealing with complex environments (system calls, pointers, external libraries), and modeling low-level behavior

Symbolic Execution

Code example to reinforce the concept.

```c
#include <stdio.h>

int classify(int x) {
    int y = x + 1;

    if (y > 0) {
        if (x % 2 == 0) {
            printf("Case A\n");
        } else {
            printf("Case B\n");
        }
    } else {
        printf("Case C\n");
    }

    return 0;
}

int main() {
    int x;
```

Symbolic Execution

There are many popular symbolic executor’s for binary program’s. We’ll be using Angr.

docs.angr.io/en/latest/

angr documentation

Welcome to angr’s documentation!

Welcome to angr’s documentation! This documentation is intended to be a guide for learning angr, as well as a reference for the API. If you’re new to angr,

The angr team maintains a number of libraries that are used as part of angr. These libraries are:

- archinfo - Information about CPU architectures
- pyvex - Python bindings to the VEX IR
- pypcode - Python bindings to the Pcode IR
- cle - Many-platform binary loader
- claripy - Solver abstraction layer

Lab 2

https://hacs408e.net/labs/week-13/lab-2/

1197

MINUTES

− +

Ends at 7:40 PM

Final

CTF challenge.