Dynamic Analysis Platform for Distributed Applications

Laurent Christophe, Coen De Roover, Elisa Gonzalez Boix

Program Comprehension (1)

```
1   function solve (coefs) {
2     const delta = coefs.b * coefs.b - 4 * coefs.a * coefs.c;
3     if (delta < 0)
4       return [];
5     if (delta === 0)
6       return [(-coefs.b) / (2 * coefs.a)];
7     return [
8       (-coefs.b - Math.sqrt(delta)) / (2 * coefs.a),
9       (-coefs.b + Math.sqrt(delta)) / (2 * coefs.a)
10    ];
11  }
12
13  const sols = solve({a:1, b:6, c:9});
14  if (sols.length === 0)
15    console.log("No Solution");
16  else if (sols.length === 1)
17    console.log("Sol = " + sols[0]);
18  else
19    console.log("Sol1 = "+sols[0]+", Sol2 = "+sols[1]);
```

```
Sol = -3
```

Instrumentation Platform

target

```
console.log("Hello!");
```

*target**

```
META.invoke(console, "log", ["Hello!"], 1);
```

weave

eval

output

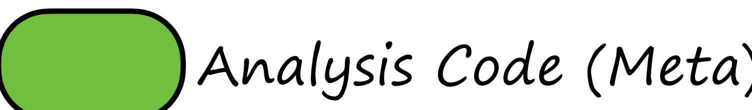
```
invoke log @1
Hello!
```

pointcut

```
["invoke"]
```

advice

```
global.META = {};
META.invoke = (obj, key, vals, ser) {
  console.log("invoke "+key+" @"+ser);
  return Reflect.apply(obj[key], obj, vals);
};
```



Simple Tracing

```
1   function solve (coefs) {
2     const delta = coefs.b * coefs.b - 4 * coefs.a * coefs.c;
3     if (delta < 0)
4       return [];
5     if (delta === 0)
6       return [(-coefs.b) / (2 * coefs.a)];
7     return [
8       (-coefs.b - Math.sqrt(delta)) / (2 * coefs.a),
9       (-coefs.b + Math.sqrt(delta)) / (2 * coefs.a)
10    ];
11  }
12
13  const sols = solve({a:1, b:6, c:9});
14  if (sols.length === 0)
15    console.log("No Solution");
16  else if (sols.length === 1)
17    console.log("Sol = " + sols[0]);
18  else
19    console.log("Sol1 = "+sols[0]+", Sol
```

```
              solve(#object) @13:13
36          = 6 * 6          @2:16
4           = 4 * 1          @2:36
36          = 4 * 9          @2:36
0           = 36 - 36        @2:16
false       = 0 < 0          @3:6
true        = 0 === 0        @5:6
-6          = -6             @6:13
2           = 2 * 1          @6:26
-3          = -6 / 2         @6:12
#array      = solve(#object) @13:13
false       = 1 === 0        @14:4
true        = 1 === 1        @16:9
"Sol = -3"  = "Sol = " + -3  @17:14
```

https://tinyurl.com/y7k3dbfu

Data-Flow Analysis

```
1   function solve (coefs) {
2     const delta = coefs.b * coefs.b - 4 * coefs.a * coefs.c;
3     if (delta < 0)
4       return [];
5     if (delta === 0)
6       return [(-coefs.b) / (2 * coefs.a)];
7     return [
8       (-coefs.b - Math.sqrt(delta)) / (2 * coefs.a),
9       (-coefs.b + Math.sqrt(delta)) / (2 * coefs.a)
10    ];
11  }
12
13  const sols = solve({a:1, b:6, c:9});
14  if (sols.length === 0)
15    console.log("No Solution");
16  else if (sols.length === 1)
17    console.log("Sol = " + sols[0]);
18  else
19    console.log("Sol1 = "+sols[0]+", Sol2 = "+sols[1]);
```

https://tinyurl.com/ybajrp34

```
#41 = primitive(1)            @/target/delta.js:13
#43 = primitive(6)            @/target/delta.js:13
#68 = unary("-", #43)         @/target/delta.js:6  // -6
#69 = primitive(2)            @/target/delta.js:6
#71 = binary("*", #69, #41)   @/target/delta.js:6  // 2
#72 = binary("/", #68, #71)   @/target/delta.js:6  // -3
#86 = primitive("Sol = ")     @/target/delta.js:17
#88 = binary("+", #86, #72)   @/target/delta.js:17 // "Sol = -3"
```

Teapless

Platform for Single-Process JavaScript

- Pretty mature
  - 3+ years old
  - 300+ commits
  - 20,000+ downloads
- Supports ES5 completely and most of ES2017
- Track primitive values across the object graph
- Supports server-side and client-side JavaScript

https://github.com/lachrist/aran

https://github.com/lachrist/linvail

https://github.com/lachrist/otiluke

Program Comprehension (2)

server.js

```
1  function solve (coefs) {
2    const delta = coefs.b * coefs.b - 4 * coefs.a * coefs.c;
3    if (delta < 0)
4      return [];
5    if (delta === 0)
6      return [(-coefs.b) / (2 * coefs.a)];
7    return [
8      (-coefs.b - Math.sqrt(delta)) / (2 * coefs.a),
9      (-coefs.b + Math.sqrt(delta)) / (2 * coefs.a)
10   ];
11 }
12 process.on("message", (message) => {
13   process.send(JSON.stringify(solve(JSON.parse(message))));
14 });
```

client.js

```
1  process.on("message", (message) => {
2    const sols = JSON.parse(message);
3    if (sols.length === 0)
4      console.log("No Solution...");
5    else if (sols.length === 1)
6      console.log("Sol = "+sols[0]);
7    else
8      console.log("Sol1 = "+sols[0]+", Sol2 = "+sols[1]);
9  });
10 process.send(JSON.stringify({a:1, b:6, c:9}));
```

Independent Per-Process Analyses

targetA → weave → targetA* → eval

pointcutB → weave

adviceB → eval

base communication

meta communication

pointcutA → weave

adviceA → eval

targetB → weave → targetB* → eval

Application Code (Base)

Analysis Code (Meta)

Generated Code

Orchestrated Cross-Process Analysis

targetA

weave

pointcut

remote advice

orchestrate

targetA*

eval

local advice

local advice

eval

targetB*

Distributed Data-Flow Analysis

server.js

```
1  function solve (coefs) {
2    const delta = coefs.b * coefs.b - 4 * coefs.a * coefs.c;
3    if (delta < 0)
4      return [];
5    if (delta === 0)
6      return [(-coefs.b) / (2 * coefs.a)];
7    return [
8      (-coefs.b - Math.sqrt(delta)) / (2 * coefs.a),
9      (-coefs.b + Math.sqrt(delta)) / (2 * coefs.a)
10   ];
11 }
12 process.on("message", (message) => {
13   process.send(JSON.stringify(solve(JSON.parse(message))));
14 });
```

client.js

```
1  process.on("message", (message) => {
2    const sols = JSON.parse(message);
3    if (sols.length === 0)
4      console.log("No Solution...");
5    else if (sols.length === 1)
6      console.log("Sol = "+sols[0]);
7    else
8      console.log("Sol1 = "+sols[0]+", Sol2 = "+sols[1]);
9  });
10 process.send(JSON.stringify({a:1, b:6, c:9}));
```

https://tinyurl.com/yc4f4714

```
#97  = primitive(1)             @/target/delta/client.js:10
#99  = primitive(6)             @/target/delta/client.js:10
#152 = unary("-", #99)          @/target/delta/server.js:6 // -6
#153 = primitive(2)             @/target/delta/server.js:6
#155 = binary("*", #153, #97)   @/target/delta/server.js:6 // 2
#156 = binary("/", #152, #155)  @/target/delta/server.js:6 // -3
#191 = primitive("Sol = ")      @/target/delta/client.js:6
#193 = binary("+", #191, #156)  @/target/delta/client.js:6 // "Sol = -3"
```

Platform for Multi-Process JavaScript

- Prototype stage
- Deploy single-process analyses in a multi-process context for free
- Capable of tracing values across process:
  - WebSocket communication between NodeJS and Browsers
  - HTML request/response between NodeJS and Browsers
  - IPC Channels between NodeJS processes
  - Message passing between WebWorkers

https://github.com/lachrist/drizzt

What can my platform mean for your company?

aran — node test/bin.js test/advice/stepper.js demo/target/delta.js — 70×29

```
{ key: 'declare',
  serial: 3,
  arguments: [ 'let', 'c', 9 ],
  estack: [ '#undefined' ],
  vstack: [ 9 ],
  cstack:
   [ { scope: [ '@global' ], frames: [ [ 'block', '&0', null ] ] } ],
  store:
   { '&0':
      { type: 'object',
        prototype: null,
        names:
         { this:
            { value: '@global',
              writable: false,
              enumerable: true,
              configurable: false },
           a:
            { value: 1,
              writable: true,
              enumerable: true,
              configurable: false },
           b:
            { value: 6,
              writable: true,
              enumerable: true,
              configurable: false } },
        symbols: [] } } }
Press <enter> to step in...
```

RECORD AND REPLAY DEBUGGING AGAINST IN-THE-WILD EXPLOITS... TURKULAINEN & NIEMELÄ

RECORD AND REPLAY DEBUGGING AGAINST IN-THE-WILD EXPLOITS AND OTHER PRACTICAL CASES

Jarkko Turkulainen & Jarno Niemelä
F-Secure, Finland

Email {jarkko.turkulainen, jarno.niemela}@f-secure.com

ABSTRACT

'Record and replay' code analysis has been a topic of interest in academia for the last 10 years, but it has not yet offered practical results when applied to exploit analysis. A 'record and replay' approach to debugging – specifically, a responsive and deterministic implementation that can record a full code execution and replay it offline, as well as step it forwards and backwards, set breakpoints and handle other common debugging tasks – would be a great advantage to exploit researchers, provided it can circumvent current anti-debugging tactics used by malware authors.

Several papers have been published on the 'record and replay' approach as it pertains to JavaScript. However, the tools described in these papers rely on specific web browsers (most commonly *Firefox*) or on specific functions in browsers. The tools' lack of support for other browsers, particularly the popular target *Internet Explorer* (and, to a lesser extent, *Edge* and *Chrome*) make them ineffective against real-life exploit kits with anti-debugging or anti-tracing capabilities. These kits are able to detect if an exploit is being run in a different browser from the expected target – for example, if exploit code for *Internet Explorer* or *Chrome* is being run in *Firefox*. In such conditions, the kit will refuse to work, based on the assumption that a researcher is trying to debug and analyse the exploit.

In our paper, we describe how we successfully apply the 'record and replay' approach to debugging, showcasing its potential benefits, and demonstrate a real-life implementation of it which can interactively debug JavaScript exploits and other malicious code. Our implementation is browser-agnostic, able to work natively in any browser being targeted by the exploit. JavaScript API manipulation is used to make it appear as though the exploit is running on a vulnerable targeted browser, leading the exploit kit to complete its attack. In reality, the malicious code is run in a secured browser where the full exploit code and execution flow is exposed without crashing the browser, allowing us to easily identify known exploits.

1. INTRODUCTION TO EXECUTION REPLAY SYSTEMS

The execution replay system was originally developed to handle issues that arise as a result of the behaviours of a non-deterministic environment. When debugging a program in such an environment, e.g. running JavaScript code on a web browser, getting consistent results for each execution poses a challenge. With the execution replay system, a program's execution can be recorded and later replayed in a consistent manner, thus allowing the developer to reproduce the problems in the program's logic deterministically.

In *A Taxonomy of Execution Replay Systems*, Cornelis *et al.* [1] presented a good analogy of this approach:

> 'If a cat tries to catch a mouse she'll first immobilize it with her paw, effectively rendering it helpless and only then go for the kill. The reason is simple: the mouse is too fast to kill when running around freely.
>
> 'The same two-phased approach is often needed during the debugging of software. The first thing that needs to be done when a bug is detected is to make sure that it can be reproduced at will. Only then can we start looking for the root cause of the bug.'

The execution replay system consists of two operational phases: recording and replaying. In the recording phase, details of the execution events are recorded onto a storage medium such as a log file or a database. Later, in the replaying phase, these records can be retrieved from the storage medium and replayed to allow further examination of the execution events in a consistent, deterministic environment.

For the system to produce useful results, Cornelis *et al.* [1] suggested that it must meet the following requirements:

- **Accuracy**: the system must be able to replicate the original execution as closely as possible.
- **Non-intrusiveness**: the recording phase must not introduce additional bugs or alter the execution environment in any way.
- **Space efficiency**: the space overhead of the storage medium must be minimized as much as possible.
- **Time efficiency**: the replayed events must be executed at a speed comparable to the original execution.

Guided by these principles, we propose an adjusted system that can be used for debugging and reverse engineering programs written in JavaScript, especially those running on unmodified browsers and operating systems.

The benefit of using an execution replay system in a reverse engineering complicated online environment is that it allows us to offload the processing of the results. It enables us to build interactive, graphical tools for analysing the records in a debugger-like environment when functions such as stepping execution backwards are not possible in a live environment.

Our research only records a limited number of execution events – to be specific, only JavaScript function calls and their arguments are recorded. By defining this limitation, we can exercise some flexibility when it comes to accuracy requirements and focus instead on the non-intrusiveness and efficiency in handling the record medium.

2. PROBLEM STATEMENT AND GOAL

The goal of our research is to create an execution replay system that would be useful in analysing modern exploit kits and other online malware. In order to accomplish this goal, the recording phase must adhere to the following basic requirements:

1. It must work on any browser and operating system, especially popular targets such as *Windows* OS and *Internet Explorer*. In the near future, the target might shift to more modern systems, such as *Edge*.
2. It must be non-intrusive, which is an important factor for two reasons:

VIRUS BULLETIN CONFERENCE OCTOBER 2017 1