Taint Analysis of JavaScript Code to Detect Web Applications Vulnerabilities

Gabriel Quadros
Conviso IT Security
gquadros@conviso.com.br
(77) 9105-0500

Summary

Summary

Topics

- Introduction
- Client-side vulnerabilities
- Approaches to analysis
- JsInstrumentator
- Conclusion

Introduction

About me

- Security Consultant and Researcher at Conviso IT Security (*www.conviso.com.br*)
- Completing the Bachelor of Computer Science at Universidade Estadual do Sudoeste da Bahia - UESB
- Involved in computer security since 2003

Introduction

Motivation

- To analyze the increasing amount of client-side code present in Web applications
- Web 2.0
- More code, more bugs
- Poorly written client-side code can also contain vulnerabilities!
- Fuzzing?!?

Introduction

Fuzzing?!?

```
<script>
    var nav = navigator.userAgent;
    if (nav.indexOf("XXX69YYY") != -1) {
        document.write("Welcome to " +
      unescape(document.location.href) +
"!");
    } else {
        document.write("Access denied!");
    }
</script>
```

Client-side vulnerabilities

JavaScript

- DOM-Based XSS
- Open Redirect
- CSRF – JSON Hijacking, …
- Session Fixation
- ...

Client-side vulnerabilities

DOM-Based XSS

```
<script>
   code = document.location.hash.slice(1);
   eval(code);
</script>
```

http://site/pagina.html#alert(1)

Client-side vulnerabilities

RIA

- XSS in Flash
- Open Redirect
- ...

Approaches to analysis

In what ways can we analyze this kind of code dinamically?

1) To analyze the IR generated by the JIT compilers
2) To analyze the Assembly code generated by the JIT compilers
3) Modify the JavaScript interpreter
4) Rewrite the JavaScript code through a Web proxy
5) Write an add-on for your browser

Approaches to analysis

1) To analyze the IR generated by the JIT compilers

- JIT compiler → Performance gain in the execution of JavaScript code
- Used by all major browsers
  - FF 3.x: *TraceMonkey/nanojit*, FF 4: *JägerMonkey/Nitro*
  - Opera >= 10.5: *Carakan*
  - Chrome: *V8* (!)
  - IE 9: *Chakra*

Approaches to analysis

1) To analyze the IR generated by the JIT compilers

- Most of them translates the JavaScript code to an IR before generating the native code
- As a result, we have an equivalent code with simpler syntax for analysis
- This approach has not been widely explored

Approaches to analysis

Nanojit – Low level intermediate representation (LIR) | *nanojit/LIRopcode.tbl*

```
125 OPDEF(feq,      27, 2, Op2)     // floating-point equality
126 OPDEF(flt,      28, 2, Op2)     // floating-point less-than
127 OPDEF(fgt,      29, 2, Op2)     // floating-point greater-than
128 OPDEF(fle,      30, 2, Op2)     // floating-point less-than-or-equal
129 OPDEF(fge,      31, 2, Op2)     // floating-point greater-than-or-equal
130
131 OPDEF(ldcb,     32, 1, Ld)      // non-volatile  8-bit load
132 OPDEF(ldcs,     33, 1, Ld)      // non-volatile 16-bit load
133 OPDEF(ldc,      34, 1, Ld)      // non-volatile 32-bit load
134
135 OPDEF(neg,      35, 1, Op1)     // integer negation
136 OPDEF(add,      36, 2, Op2)     // integer addition
137 OPDEF(sub,      37, 2, Op2)     // integer subtraction
138 OPDEF(mul,      38, 2, Op2)     // integer multiplication
139 OPDEF(div,      39, 2, Op2)     // integer division
140 OPDEF(mod,      40, 1, Op1)     // hack: get the modulus from a LIR_div result,
141
142 OPDEF(and,      41, 2, Op2)     // 32-bit bitwise AND
```

Approaches to analysis

1) To analyze the IR generated by the JIT compilers

- Advantages
  - Ease of analysis: three-address code, SSA, etc.
- Disadvantages
  - JIT compiler dependent
  - The analysis may be incomplete: the compiler supports DOM objects? The compilation occurs only when it detects a *hot spot*?
  - Installation

Approaches to analysis

2) To analyze the Assembly code generated by the JIT compilers

- Directly analyze the native code
- V8 engine
- Has also not been widely explored

Approaches to analysis

2) To analyze the Assembly code generated by the JIT compilers

- Advantages
  - Maybe reuse of analysis tools written for native code: Valgrind and PIN plugins, VINE+TEMU, BAP, etc.
- Disadvantages
  - Browser dependent: you need to know where to find the native code generated and how the DOM objects are represented
  - The analysis may be incomplete
  - Installation

Approaches to analysis

3) Modify the JavaScript interpreter

- Choose a browser that uses the desired interpreter, which should be preferably Open Source
- Locate the code responsible for interpreting the JavaScript and modify it
- How to modify the interpreter?
  - Insert the analysis code in the interpretation code
  - Add code to generate *run traces* in the interpreter's IR and then analyze
  - Add code to generate *run traces* in your own IR and then analyze

Approaches to analysis

3) Modify the JavaScript interpreter

- Advantages
  - Direct implementation: the interpreter is usually structured as a big *switch-case* structure
- Disadvantages
  - Browser dependent
  - Restriction of the language used in development: typically C or C++
  - Installation

Approaches to analysis

4) Rewrite the JavaScript code through a Web proxy

- Use a Web proxy to intercept the JavaScript code and rewrite it to add the analysis code
- Can be implemented without a Web proxy, by modifying the browser to intercept the code
- As the previous method, is widely used in academic research

Approaches to analysis

4) Rewrite the JavaScript code through a Web proxy

- Advantages
  - Browser independent
  - Ease of development
  - Installation
- Disadvantages
  - Changes the original JavaScript code

Approaches to analysis

5) Write an add-on for your browser

- Used in some community projects. Ex.: *noXSS*
- It can be interesting if the API provides support to JavaScript analysis

Approaches to analysis

5) Write an add-on for your browser

- Advantages
  - Installation
- Disadvantages
  - Browser dependent
  - Subject to the limitations of the API

JsInstrumentator

Introduction

- A Web proxy to perform dynamic analysis of JavaScript code
- Detection of vulnerabilities that involve the use of user-controlled data in *dangerous* methods like: *eval()*, *document.write()*, etc.
- Taint Analysis over strings
- Available at: http://code.google.com/p/jsinstrumentator/

JsInstrumentator

Taint Analysis

- Information Flow Analysis
- “Information flows from object *x* to object *y*, denoted *x* => *y*, whenever information stored in *x* is transferred to, or used to derive information transferred to, *y*.” (Denning)
- Based on the work “Detecting History sniffing via Information Flow” of Jang et al., available at: http://*pho.ucsd.edu/rjhala/dif.pdf*

JsInstrumentator

Taint Analysis – Steps

- Define the sources of untrusted, user-controllabe data
- Define the critical points where a tainted data should go to detect a vulnerability
- Propagate the tainted data
  - To taint an object, we add a taint mark which allow us to track the propagation of the initial sources

JsInstrumentator

Taint Analysis – Some untrusted data sources

document.URL

document.URLUnencoded

document.location.*

document.cookie

document.referer.*

window.location.*

forms.value

JsInstrumentator

Taint Analysis – Some critical points

eval(), window.execScript(), window.setInterval(), window.setTimeout()

document.write(), document.writeln(), document.body.innerHtml=

document.forms[0].action=, document.attachEvent(), document.create()

document.execCommand(), window.attachEvent()

document.URL=, document.location=, document.open(), window.open()

document.cookie=

JsInstrumentator

How is the analysis performed?

- Rewriting the JavaScript code
- Need to parse the JavaScript AST
- Rewriting rules

JsInstrumentator

Example – Original code

```
<script>
  a = “123”;
  b = document.location;
  c = a + b;
  document.XXX.innerHTML = c;
</script>
```

JsInstrumentator

Example – Rewritten code

```
// TSET object
```

```
<script>
  // a = “123”;
  (TSET.direct.push(),
  tmp1 = “123”,
  tmp2 = TSET.taint(tmp1),
  TSET.check(tmp2, “a”),
  a = tmp2,
  TSET.direct.pop(),
  tmp2)
```

JsInstrumentator

```
// b = document.location;
(TSET.direct.push(),
tmp1 = document.location,
tmp2 = TSET.taint(tmp1),
TSET.check(tmp2, “b”),
b = tmp2,
TSET.direct.pop(),
tmp2)
```

JsInstrumentator

```
// c = a + b;
(TSET.direct.push(),
    tmp1 =
        (tmp2 = a,
         TSET.direct.add(tmp2),
         tmp3 = b,
         TSET.direct.add(tmp3),
         tmp4 = a + b,
         tmp4 = TSET.taint(tmp4),
         tmp4
        ),
    tmp5 = TSET.taint(tmp1),
TSET.check(tmp5, “c”),
c = tmp5,
TSET.direct.pop(),
tmp5)
```

JsInstrumentator

```
// document.XXX.innerHTML = c;

    ...
    TSET.check(..., “document”, “XXX.innerHTML”),
    ...
```

JsInstrumentator

Implementation

- Python
- Twisted Web for the Web proxy
- BeautifulSoup for parsing HTML
- Pynarcisus for parsing JavaScript
- Integration with Firebug to detect vulnerabilities

JsInstrumentator

Next steps

- Extend the Taint Analysis for other data types
- Add support for detecting other types of vulnerabilities
- Integration with a *string solver* to improve fuzzing
- Community contribution

Conclusion

Conclusion

Conclusions

- To perform more advanced analyses of client-side code is a real need
- The approaches presented can be applied to other file formats which can hold code
- It can also be used to protect against the exploitation of vulnerabilities

Questions?