ShiftLeft

Static Analysis of a Dynamic Language

JavaScript is the most widely used programming language for the client-side of web applications, powering a majority of today's websites. Its use, however, extends to the server-side via Node.js. Given its ubiquity, how do developers ensure the security of their applications?

One method by which application developers can ensure the security of their applications is with the use of **static analysis**, either during the development process or prior to deployment. Because JavaScript is an interpreted language that has no notion of compile-time, the runtime must check every operation to ensure its safety. That said, static analysis of JavaScript can be challenging.

Why It is Difficult to Analyze and Detect Bugs in JavaScript Applications

Static analysis of JavaScript applications is a challenging task. JavaScript is a dynamic and weakly typed language with first-class functions. It has objects, but no classes. It relies on prototyping instead of inheritance to share and extend functionality. JavaScript's prototyping functionality also makes it an expressive scripting language, but at the same time, this exposes an environment in which runtime errors can occur.

Challenge #1

In JavaScript, user-defined objects can be created without a class definition. Furthermore, variables and functions can be dynamically added to objects

The following script demonstrates JavaScript's classless and dynamic properties, and it is these two features that make static type checking of JavaScript difficult:

```
> var person = { name: "Adam", toString: function() { return this.name }}
> person.toString()
Adam

> person.wife = "Alice"
Alice

> person.setWife = function(b) { this.wife = b}
function (b) { this.wife = b; }

> person.setWife("Eve")
> person.wife

Eve
```

What happens through the script is as follows:

1. The script creates a person object with a field that specifies the person's name and a function that returns the name
2. After instantiating the object and printing the name, a new field is added to the person, specifying the person's wife
3. Finally, a mutator for the wife field is added to the object; the mutator is invoked, setting the wife field to a new value

Challenge #2

JavaScript is a loosely typed language that allows you to define variables using the **var** keyword, which places no restrictions on the type of variable you're creating.

```
> function add(a, b) { return a + b }
> function add() { return arguments[0] + arguments[1] }

> add(1, 0) // pass Number type
1

> add("Hello ", "World") // pass String type
Hello World

> add() // pass no arguments
NaN

> add(1,2,0) // pass more arguments than the number supported
3
```

Because you can define a variable as one type, then later use it as another type, JavaScript using the inferred type of variables to determine the correct operations to perform (i.e., it needs to determine whether the plus **(+)** symbol is intended to add two numbers or to concatenate two strings).

Challenge #3

JavaScript allows the modification of a variable's type based on the context of its usage.

```
> typeof 0
number

> typeof true
boolean

> typeof "string"
string

> typeof (new Array())
object

> typeof function f() {}
function

> var x = 1
> x = !!x
true

> x += "string"
truestring

> x = 10 + !!x
11

> x = "40" + "1" - x
390
```

In the script immediately above, you can see a variable changing its type from number > Boolean > string > Boolean > integer. The final example shows how two strings can be concatenated, then converted to an integer for subtraction.

Variable Typing and Vulnerabilities in JavaScript Applications

The ability to dynamically add and update members to objects in JavaScript results in many actions that result in runtime errors, including:

- Applying an arithmetic operator to an object that is not a wrapper object for a number
- Accessing a variable that doesn't exist
- Accessing the properties of a null object

Runtime Errors

Many of the runtime errors that occur are due to invalid type conversions (which is partially due to the lack of classes in JavaScript). Type conversion errors occur when arithmetic operations are performed on non-numeric values. If operands cannot be cast to numeric types, a runtime error is thrown.

For example, subtracting two strings results in an invalid number:

```
> “hello “ - “world”
NaN
```

Similarly, dividing two strings is an invalid operation when dividing two strings (note that the second string is non-numeric):

```
> “5” / “one”
NaN
```

Type conversions are unpredictable; even providing Boolean values (though the first is provided in the form of a string) is no guarantee and the result returned is an integer

```
> “true” & false
0
```

The following is another example of unexpected behavior; instead of of 1, the result is NaN:

```
> “true” - false
NaN
```

Many of these errors could have been flagged before execution using a type-checking tool, resulting in fewer runtime errors. The one caveat, however, is that the number of formal parameters in a function definition doesn’t need to match the number of actual parameters; if there are too few parameters, one can assume that the outstanding parameters are undefined.

Prototype-Based Errors

JavaScript is a prototype-based language supporting the dynamic addition of members to objects. Runtime errors are therefore expected whenever a script references an undefined variable or a non-existent member of an object.

```
> var obj = { x:1 }
> print(obj.x)
1

> print(obj.y)
undefined

> obj[“undefined”] = “should be undefined”
> print(obj[obj.y])
should be undefined
```

The code snippet above shows a prototype-based error in JavaScript resulting from the following actions:

1. Create an object with one member variable, then print
2. Attempt to access member variable y, which is undefined
3. Modify the value of undefined (which is use incorrectly)

However, despite being a classless programming language, one of the most commonly found idioms in contemporary JavaScript applications is the emulation of class-based, object orientation using the prototype system.

Types of Vulnerabilities Due to Weak Typing in JavaScript

The lack of type safety in JavaScript presents security vulnerabilities, including:

- Clickjacking and phishing by mixing layers and iframe
- CSRF and leveraging CORS to bypass SOP
- Attacking WebSQL and client-side SQL injection
- Information theft from local/cookie storage and global variables
- Cross-Site Scripting (XSS) and HTML 5 tag abuse
- HTML 5/DOM-based XSS and redirects
- DOM injections and hijacking with HTML 5
- The abuse of thick client features

- The use of WebSockets for stealth attacks
- The abuse of WebWorker functionality
- The use of components with known vulnerabilities

The Use of JavaScript on the Server-Side

One of the biggest contributors to the use of JavaScript on the server-side is Node.js, an event-based network application platform that utilizes asynchronous programming interfaces for I/O operations. JavaScript is the language of development when using Node.js.

Using JavaScript in domains other than the client-side allows front-end developers to work in other environments. However, this skill transfer may raise the risk of introducing vulnerabilities and types of attacks that were previously not a problem, since they are risky only on the server-side and have no client-side corollary.

Node.js and the Use of Third-Party Libraries

A characteristic that's common to many Node.js applications is the extensive use of third-party libraries. The npm platform alone offers over 1 million packages (most of which are libraries), and only a few have been screened for security vulnerabilities.

One of the challenges when analyzing the security of npm packages is that they are often not self-contained; rather, they depend on other npm packages for lower-level functionality. On average, an npm package depends on seventy-nine (79) other packages and on code that's published by thirty-nine (39) developers. To fully understand an application using npm packages, you need to consider all of these dependencies as well.

Security Tools for Third-Party Libraries

There are tools that aggregate known security vulnerabilities in specific versions of individual libraries and report them to developers. For example, npm audit analyzes all of the dependencies in a Node.js application and warns the developer about any known vulnerabilities in the code on which the application depends.

The downside to this approach is that the developer has a limited view that leads to a high number of false positives. Often, the primary part of the library isn't used by the application or it is used in a way where a security vulnerability isn't likely to affect the application's overall security.

For example, an application might use an npm module that's vulnerable to command injection attacks, but the application only passes string constants provided by the developer as input to the module. It is important to make a distinction between relying on a library that contains a potential known vulnerability and using the library in an insecure way.

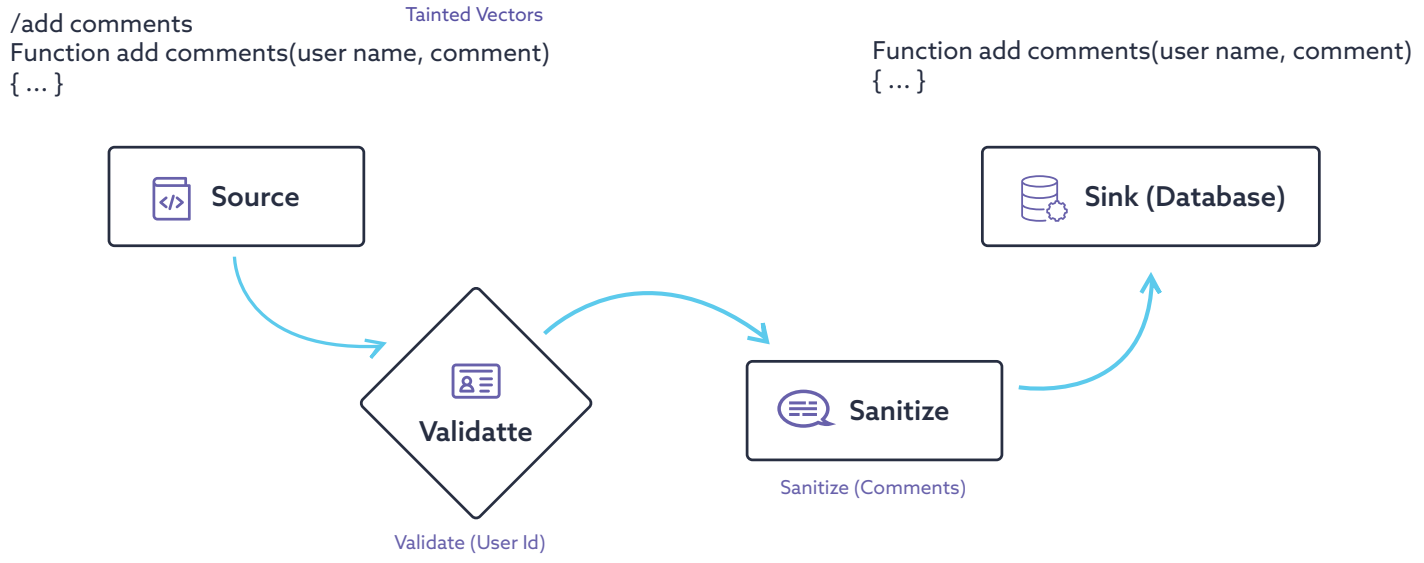
Another downside is that such security tools miss libraries that use insecure features inherent in JavaScript or Node.js; these issues are often not registered as known vulnerabilities if the documentation indicates that these features are used internally.

Detecting Security Issues in JavaScript Applications

Given the ubiquity of JavaScript, it's important to find the security vulnerabilities introduced whenever a developer uses its dynamic features.

Because websites typically contain both user interaction features and third-party components, the latter (if untrusted) may be used to exploit vulnerabilities. For example, a malicious party might inject an input string value that can be parsed into executable JavaScript code that, if not sanitized, can lead to a cross-site scripting attack.

Many security problems can be formalized as information flow problems that seek to preserve the integrity of the data (that is, untrusted values can't affect a sensitive value or operation) and the confidentiality of the data (that is, sensitive values cannot be observed from outside the computation)



To help mitigate security issues, researchers and industry practitioners recommend taint analysis, which detects the flows of data that violate program integrity and data confidentiality, as a more precise approach for securing JavaScript applications.

Type Checking During Compilation

Using a tool to help with type checking during compilation can help us enforce better practices and reduce the likelihood of vulnerabilities. Such type checking tools typically rely on static analysis to approximate the program's behavior, with call graphs being one of the more commonly used program representations.

Call graphs associate each call site in a program with the set of security-sensitive functions that could be invoked from that site. For example, a tool for finding security vulnerabilities might use a call graph to detect possible data flows from tainted inputs to security-sensitive functions.

A traditional call graph is a directed graph that connects call sites with call targets (e.g., function declarations).

Call graphs are useful for taint-flow analysis, debugging, refactoring, and many other applications. For languages with polymorphism or higher-order functions, the call graph isn't available from the source code; rather, it must be statically approximated (or what we call the fuzzy approach).

Call graphs help answer two crucial questions:

1. What calls a function?
2. What does a function call?

There are several challenges inherent to such an approach, however.

Challenge #1: Values

A basic part of data flow analysis is to model the values that can appear at runtime. JavaScript is dynamically typed, which means that the analysis must be able to handle the fact that a specific variable can contain values of different types (e.g., string and integer) at different points during execution. The analysis must be able to recognize conversions, both implicit and explicit, during execution. To be useful for analysis, we must track all such possibilities.

Challenge #2: Control Flow

JavaScript has several features that make control flow analysis complicated, such as higher-order functions (which mean that data flow and control flow are intertwined since you can pass functions as values), as well as the throwing and catching of exceptions, which lead to alternative paths of execution that must be tracked. There are instances where control flow and data flows cannot be analyzed separately.

Challenge #3: Asynchronous Event Handling

Traditional call graph-based approaches reflect only the interprocedural flow of control due to function calls while ignoring the event-driven flow of control commonly found in JavaScript applications.

Node.js applications are typically written in an event-driven style that relies heavily on callbacks invoked when an asynchronously executed operation completes. This method of application development gives rise to several new types of bugs. For example, an application may emit events for which no listener has been registered yet, or an event listener may be unreachable because the event name was misspelled or the listener was registered on the wrong object.

However, in languages with asynchronous callbacks and event listeners, a traditional call graph provides incomplete information because the call graph does not show how events result in indirect calls.

In such cases, the code analysis engine should be able to determine that no such predetermined ordering exists in the context of an asynchronous handler from a data-tracking perspective. In an event-based system, information about the program's control flow isn't immediately available from the program's syntax, and events that occur may cause the program to switch between event listeners at certain points during execution.

Some of the issues related to event-based programs include:

- **Dead Emits:** a dead emit occurs when an emit expression doesn't cause an event listener to be scheduled. Typically, this is caused when the wrong event emits or when the event is emitted onto the wrong object. The Node.js API, unfortunately, makes these all the more common with its use of similar-sounding names (e.g., connect and connection)
- **Dead Listeners:** a dead listener occurs when an event listener is registered on an object for an event, but the event is never emitted after registration. The event listener may be registered on the wrong object or for the wrong event. A program may have a dead listener independently of a dead emit
- **Data Race:** a data race occurs when two event listeners communicate via a shared state. For example, one event listener may write a piece of the global state, which is then read by the second event listener. If the write must happen before the read can succeed, then there must be a may-happen-before relationship between the write and read

A static analysis engine should be able to reason through events and event listeners to detect bugs that may become security vulnerabilities.

Adequate Program Representation

To carry out static analysis, there needs to be an adequate and appropriate representation of a program. Furthermore, to carry out data flow-sensitive analysis, the representation must reflect the order of the program's instructions.

To that end, the source code of a program can be represented as a set of graphs, with one graph for each function present. The presence of higher-order functions means that it's unclear which functions will be invoked at a given call site, so the analysis process is responsible for the insertion of interprocedural edges. The nodes in the graph are blocks, with each block representing a sequential set of instructions in the original program with one entry point (source) and several exit points (sinks).

Interprocedural vs. Intraprocedural Analysis

When analyzing programs consisting of multiple functions, there are two options: interprocedural analysis and intraprocedural analysis. With the latter, the analysis does not propagate flow across function the function boundaries, and all information that's extracted is local to the current function.

Interprocedural analysis, however, involves propagating information across function invocations. Control and data flow are interdependent in JavaScript, so it's unknown which function a given invocation will be called prior to the beginning of the analysis. Therefore, special interprocedural edges are added during analysis as data flow facts are discovered; for each function, the analysis adds a call edge and a return edge.

However, naively adding interprocedural edges leads to an unacceptable loss of precision.

The Need for Context Sensitivity

To combat infeasible flow and the imprecision, context-sensitivity needs to be added. When context-sensitivity is enabled, a finite number of contexts exist in the analysis, and each function is analyzed in a specific context with different flows originated from different contexts kept separate.

JavaScript offers many context choices as an object-oriented language since many functions are associated with a variety of objects, strings, etc. Furthermore, JavaScript's lack of formal semantics means that designing a sound analysis program will always be a best-effort enterprise -- no proof of correctness can be given.

A traditional dense analysis propagates data flow facts along the edges of the control flow graph. A forwards analysis propagates data flow facts from a control flow graph node to its successors, while a backward analysis propagates data flow facts to its predecessors.

Every program point, represented by a control flow graph node, maintains an entire abstract state. Immutable data structures can reduce some of the memory overhead of dense data flow analysis, but these techniques are not always applicable nor sufficient, especially when dealing with dynamic languages.

How the Code Property Graph Enables Accurate and Efficient Analysis of Applications Written Using Dynamic Languages

The code property graph is a concept based on the following observation: there are many different graph representations of code, and patterns in code can be expressed as patterns in the graphs. While all of these graphs represent the same code, some properties may be easier to express using one representation instead of another.

With that in mind, different parts of the program's code can be represented with different graphs. Then, all such graph representations can be merged to get the value-add of each individual graph.

Finally, the resulting representation as a property graph, which is the native storage format of graph databases, can be expressed, enabling the expression of patterns via graph-database queries.

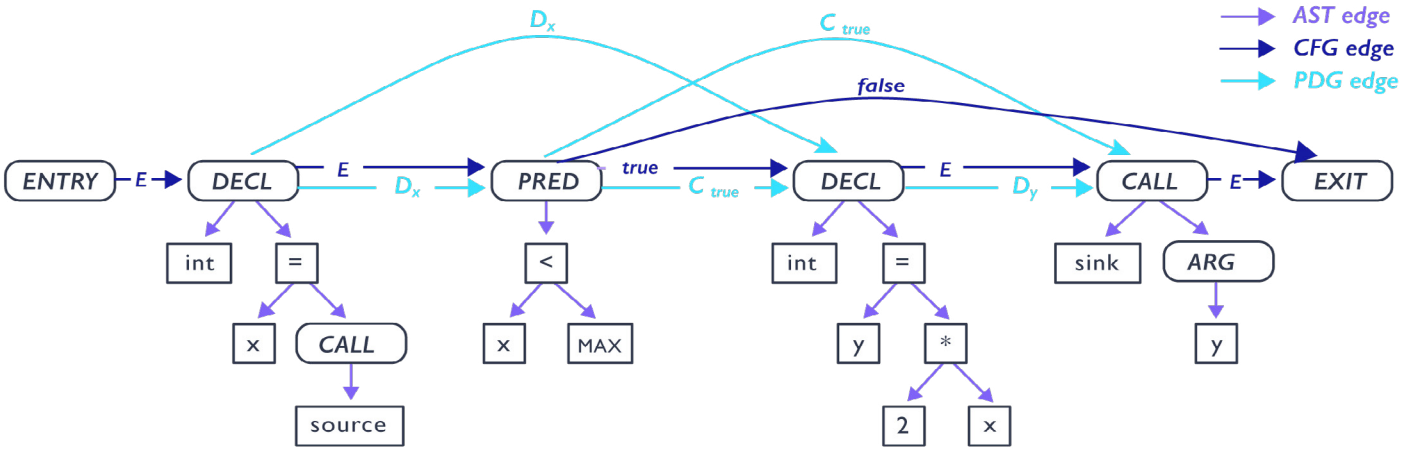


Illustration of a code property graph from the original paper "Modeling and Discovering Vulnerabilities with Code Property Graphs", where an abstract syntax tree, control-flow graph, and program-dependence graph are merged to obtain a representation for querying code

This idea, originally published by Nico Golde, Daniel Arp, Konrad Rieck, and Fabian Yamaguchi at Security and Privacy in 2014, was extended for use with interprocedural analysis one year later. The definition of the code property graph is fairly flexible; the only requirement is that certain structures must be merged while leaving the graph schema open.

It is a presentation of a concept (data structures), along with basic algorithms for querying it to uncover flaws in a program. Graphs are manually created for the targeted programming language; as such, concrete implementations of code property graphs may differ substantially from one to another.

The code property graph enables the efficient switching between intra- and interprocedural data flow analysis, giving precise, context-sensitive results. These results improve the call graph, which in turn improves the data flow tracking.

Furthermore, the code property graph enables us to create an ad hoc, approximated type system that greatly reduces the amount of individual call site resolutions and is thus key to analyzing ECMAScript-compliant code in a reasonable amount of time.

Conclusion

Applying static analysis during the software development life cycle to your JavaScript application will help ensure its safety. However, static analysis of such applications is a challenging task, but the use of the code property graph allows us to implement efficient, accurate scans.