Demo: Detecting Third-Party Library Problems with Combined Program Analysis

Grigoris Ntousakis
TU Crete
gntousakis@isc.tuc.gr

Sotiris Ioannidis
TU Crete
sotiris@ece.tuc.gr

Nikos Vasilakis
CSAIL, MIT
nikos@vasilak.is

ABSTRACT

Third-party libraries ease the software development process and thus have become an integral part of modern software engineering. Unfortunately, they are not usually vetted by human developers and thus are often responsible for introducing bugs, vulnerabilities, or attacks to programs that will eventually reach end-users. In this demonstration, we present a combined static and dynamic program analysis for inferring and enforcing third-party library permissions in server-side JavaScript. This analysis is centered around a `RWX` permission system across library boundaries. We demonstrate that our tools can detect zero-day vulnerabilities injected into popular libraries and often missed by state-of-the-art tools such as `snyk test` and `npm audit`.

CCS CONCEPTS

• **Software and its engineering** → *Automated static analysis*; Dynamic analysis; *Scripting languages*; • **Security and privacy** → **Software and application security**.

KEYWORDS

Dynamic Program Analysis, Static Program Analysis

ACM Reference Format:
Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. 2021. Demo: Detecting Third-Party Library Problems with Combined Program Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3460120.3485351

1 INTRODUCTION

Modern software development relies heavily on third-party libraries. Applications use several dozens or even hundreds of libraries, created by many different authors and accessed via public repositories. The heavy use of libraries is particularly common in JavaScript applications [6, 8, 12, 13, 15], and especially in those running on the Node.js platform [16, 19], where developers have millions of libraries at their fingertips through the npm package manager.

Security Problems: Reliance on libraries introduces several security risks—ranging from dynamic compromise, the runtime exploitation of a benign library via its inputs, to full-fledged malicious library operation—affecting the security of the entire application and its broader operating environment. For example, consider a (de)serialization library that uses JavaScript's built-in `eval` function to parse a string into a runtime object. While the library itself is benign, accessing no other external API apart from `eval`, an attacker may pass a malicious serialized object to the deserialization function, which in turn will pass it to `eval`. As a result, the library may be subverted into malicious behavior, e.g., accessing the file system or the network, that goes far beyond what a (de)serialization library is supposed to do. The underlying problem is that every library running on Node.js has all privileges offered by the JavaScript language and its runtime environment. In particular, each library is allowed to access any built-in API, global variables, APIs of other imported libraries, and even import additional libraries.

Overview: In this demo, we show how to leverage a combined static and dynamic program analysis to understand program behavior prior to the program's production execution and enforce this behavior during the program's production execution. Our techniques form a sharp contrast to state-of-the-art vulnerability detection tools such as npm audit [9] and snyk [11]: while these tools scan a program's dependencies to report on *known* attacks—collected from vulnerability reports accessible publicly—our tools can detect and notify developers of previously unseen, zero-day attacks, as we show during the demonstration of our tools.

Demo Outline: The demonstration starts by exemplifying the use of third-party libraries common in server-side Node.js development today. It then shows the expected (normal and benign) behavior of these libraries as part of larger applications, and then demonstrates unexpected (abnormal and malicious) behavior of these libraries when subverted by attackers—for example, an attacker can read and exfiltrate the contents of `/etc/passwd`. It then applies state-of-the-art vulnerability detection tools such as `npm audit` and `snyk test`, which do not report any risks—due to the reason that both tools report only known vulnerabilities. The demo finally demonstrates the use of a combined program analysis designed to report on the permissions used by third-party libraries—showing the set of permissions required for the normal operation of a library, and thus delineating between normal and malicious operation. *All the tools presented in this demonstration are open-source software.*

2 RELATED WORK

This section briefly outlines static and dynamic analysis techniques.

Static analysis: Static program analysis is a technique for extracting information about the behavior of a program by inspecting its source code. Static analysis tends to focus on invariants related to *all* executions of the program, but often misses information related

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3485351

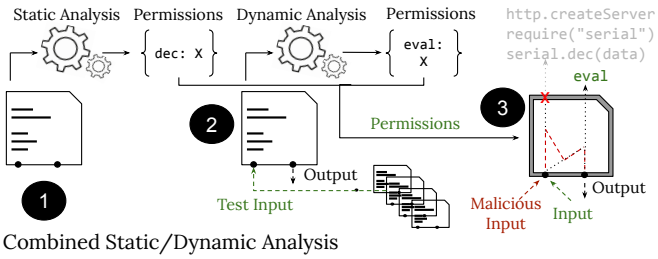

Figure 1: A static analysis tool extracts the static permissions from a Node.js third-party library. The dynamic analysis tool then extracts the corresponding dynamic permissions using the test cases. The combination of permissions are used as input to the policy enforcement component.

to dynamic program behavior. Several static analysis systems have been developed for Node.js [5, 7, 18].

Dynamic analysis: Dynamic program analysis is a technique for extracting information about a program by instrumenting its execution. Because of its nature, dynamic analysis can extract a wealth of information about a single execution but (1) this information might not generalize to other executions, and (2) it might impose a significant runtime overhead to the program's execution. Several dynamic analysis frameworks have been developed for JavaScript [2, 10, 14, 17].

Combined analysis: While both static and dynamic analysis are necessarily imprecise approximations of program behavior, their relative trade-offs make them complementary tools in a programmer's tool arsenal [3, 4]. By combining these two synergistic approaches, as our demo shows, we aim at providing improved analysis results with minimal-to-zero developer effort.

3 TOOL OVERVIEW

Fig. 1 shows an overview of our proposed techniques and the way they are applied on a real use case.

Our techniques start by running a static program analysis on the source code of the target library to extract a first set of candidate permissions (Fig. 1, (1)). This phase analyzes the source code of the library and corresponding dependencies to extract the set of interfaces—e.g., functions, global objects, language built-ins—used by the library.

Our techniques then pair this static permission set with a second set gathered via dynamic program analysis (Fig. 1, (2)). During this phase, dynamic analysis is applied against the testing infrastructure of the library, which encodes anticipated library behaviors envisioned by the library's developers. We augment these test inputs with ones gathered via active learning [15]—a critical addition for libraries that do not have test cases.

Finally, our techniques enforce the permission sets gathered by both analysis phases by instrumenting program execution (Fig. 1, (3)). When this enforcement instrumentation framework detects an access outside of the generated permission set, it throws an exception aborting the execution of the program.

4 A REAL EXAMPLE

This section exemplifies our techniques against dynamic library subversion—a common attack vector in libraries that evaluate user input.

A de-serialization library: Consider a Node.js application that uses a third-party (de)serialization library for converting serialized strings into in-memory objects. The (de)serialization library is fed client-generated strings, which may lead to remote code execution (RCE) attacks. The code below shows the relevant application fragment:

```
const serial = require('serialization');
http.createServer((req, res) => {
  req.on('end', () => {
    let val = serial.deserialization(data);
    if (val.token == 'a1b2c33d4e5f6g7h8i9jakblc')
      console.log('Api key:', val) });
```

The code above first imports the `serialization` library. It then creates a web server that receives user-provided values arriving from the network as *strings*, which get deserialized into in-memory objects. Values containing a special token are printed in the console.

Unfortunately, this deserialization functionality is provided by `serial.dec` which is implemented by a third-party library developed by programmers other than the application's nominal developers. Internally, this function uses the unsafe `eval` primitive of Node.js which evaluates *any* valid JavaScript code:

```
module.exports = {
  dec: (str) => {
    let obj;
    obj = eval(str);
    return obj;  } }
```

Benign vs. malicious operation: Benign user requests work as expected—*e.g.*, , the following request will cause the value to be printed:

```
let key = 'a1b2c33d4e5f6g7h8i9jakblc');
request.write(payload); // part of a request
```

However, adversaries can pass Turing-complete programs that will execute on the host environment—*e.g.*, , the following input will create a file `pwned.txt` using the `fs` library of Node.js:

```
let payload = 'require("fs").
    writeFileSync("./pwned.txt","uh-oh!\\n")');
request.write(payload);
```

Applying state-of-the-art tools: We attempt to detect this malicious operation using two state-of-art tools, Snyk [11] and NPM audit [9]. Running `snyk test` in the folder that contains the vulnerable library does not report any vulnerabilities:

```
Tested 1 dependencies for known issues,
no vulnerable paths found
```

The results are similar for `npm audit`:

```
found 0 vulnerabilities in 1 scanned packages
```

The reason these tools fail to report any risks is that the dependencies of our program do not have any known vulnerabilities.

Applying Static Analysis: We first run `perm.js -s`, our static permission inference analysis, to extract the first set of permissions for `serialization`:

```
{ "~/libs/serialization/index.js":
  { "eval": "rx",
    "module": "r",
    "module.exports": "w"  } }
```

The inferred permissions show the use of `eval` and `module.export` for evaluating code and exporting library functionality.

Applying dynamic analysis: We then run `perm.js -d`, our dynamic permission inference analysis, with the use of the provided test cases in order to extract permissions from the third-party library `serialization`. As all of the inputs are JSON objects, they only additional permissions are related to a few built-in primitives such as the `Array` constructor and the value `null`.

```
{ "~/libs/serialization/index.js":
  { "eval": "rx",
    "module": "r",
    "module.exports": "w",
    "Array": "rx",
    "null": "r",  } }
```

Executing with permission enforcement: We launch the instrumented program enforcing the combined `RWX` permissions inferred during the previous two phases. When the malicious input attempts to access the `fs` library, the instrumented code throws an exception that halts the execution of the program.

5 DISCUSSION & CONCLUSION

We hope that our demo will form the basis of a discussion around the practices of third-party libraries. We outline a few potential discussion threads below.

First, what is the best way for developers to incorporate specific standards to the libraries they develop and make available to the community? The goal here is to minimize supply-chain attacks due to developer mistakes. A formalization could include language-specific standards (e.g., minimizing the use of `eval`, or enforcing the inclusion of test cases with adequate coverage) for ameliorating security problems before the libraries are shared.

Second, what are the possible steps to be taken by library repositories in order to shield the community against these problems? The goal here is to identify a set of simple steps that repositories can take in order to mitigate many of these problems—with minimal overhead for the end user.

Finally, what is a good way to improve checks on program updates? As the SolarWinds [1] attack has demonstrated, discovering and mitigating vulnerabilities related to program updates is of paramount importance, and thus automating these checks to the extent possible would provide significant security benefits.

We hope that our demonstration of a combined static and dynamic program analysis in the context of real Node.js applications, will serve to kick off a targeted discussion around the problems of third-party libraries and possible ways to mitigate them.

ACKNOWLEDGMENTS

This work was partly supported by DARPA contract no. HR0011202-0013, HR001120C0191, and HR001120C0155. This work has also received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927 (CONCORDIA) and under grant agreement No 952690 (CYRENE).

REFERENCES

[1] 2020. CVE-2020-10148. Available from NIST, CVE-ID CVE-2020-10148.. https://nvd.nist.gov/vuln/detail/CVE-2020-10148

[2] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.* 50, 5 (2017), 66:1–66:36. https://doi.org/10.1145/3106739

[3] Michael D Ernst. 2003. Static and dynamic analysis: Synergy and duality.

[4] Chris Hawblitzel and Thorsten Von Eicken. 1998. *A case for language-based protection.* Technical Report. Cornell University.

[5] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020).*

[6] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. (2017).

[7] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static analysis of event-driven Node. js JavaScript applications. *ACM SIGPLAN Notices* 50, 10 (2015), 505–519.

[8] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security.* 736–747.

[9] npm. 2016. Run a security audit. https://docs.npmjs.com/cli/v7/commands/npm-audit/. https://docs.npmjs.com/cli/v7/commands/npm-audit

[10] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013).* ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[11] Snyk. 2016. Find, fix and monitor for known vulnerabilities in Node.js and Ruby packages. https://snyk.io/. https://snyk.io/

[12] Deian Stefan. 2015. *Principled and Practical Web Application Security.* Stanford University.

[13] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. 2014. Protecting Users by Confining JavaScript with {COWL}. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14).* 131–146.

[14] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient Dynamic Analysis for Node.Js. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018).* ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/3178372.3179527

[15] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21).* Association for Computing Machinery, New York, NY, USA, 16. https://doi.org/10.1145/3460120.3484736

[16] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Networked and Distributed Systems Security (NDSS'18).* https://doi.org/10.14722/ndss.2018.23131

[17] Nikos Vasilakis, Grigoris Ntousakis, Veit Heller, and Martin C. Rinard. 2021. Efficient Module-Level Dynamic Analysis for Dynamic Languages with Module Recontextualization. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021).* Association for Computing Machinery, New York, NY, USA, 1202–1213. https://doi.org/10.1145/3468264.3468574

[18] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21).* Association for Computing Machinery, New York, NY, USA, 18. https://doi.org/10.1145/3460120.3484535

[19] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Smallworld with High Risks: A Study of Security Threats in the Npm Ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19).* USENIX Association, USA, 995–1010.