

Predicting the Unpredictable: Cracking Web Randomness

Arnav Vora
University of California, Los Angeles
arnavvora@ucla.edu

Alexander Edwards
University of California, Los Angeles
aledwards25@ucla.edu

Carlos Martinez
University of California, Los Angeles
seamar@ucla.edu

Andrew Kuai
University of California, Los Angeles
andrewkuai628@ucla.edu

Shannon Wang
University of California, Los Angeles
shannonwang@ucla.edu

Anthony Fangqing Yu
University of California, Los Angeles
anthonyfangqing@ucla.edu

Abstract—Insecure pseudo-random number generation (PRNG) mechanisms, notably within JavaScript environments, may pose significant vulnerabilities in web applications if used incorrectly. This project examines the XorShift128+ PRNG, as employed in V8 JavaScript, exposing its susceptibility to attacks through reverse engineering techniques that allow for prediction of past and future outputs. Using tools such as the Z3 theorem prover, we explore methods to deduce PRNG states from observed outputs, thereby enabling potential exploits. Additionally, we analyze the "single-packet attack" vector within HTTP/2, leveraging its ability to batch HTTP requests in TCP packets to compel atomic processing of requests. Taint analysis tools can help facilitate the tracking of PRNG-based data flows to assist in the development of exploits. By combining these methodologies—PRNG state prediction, HTTP request control, and taint analysis—our research demonstrates a comprehensive exploit model applicable to vulnerable web applications, offering insights into the critical security implications of web randomness.

Index Terms—Pseudo-random Number Generator, PRNG, XorShift128+, HTTP/2, Single Packet Attack, Taint Analysis, Semgrep, CodeQL, JavaScript, V8

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

I. INTRODUCTION

Today, web applications rely heavily on JavaScript's pseudorandom number generators (PRNGs) to manage various tasks, from session identifiers to elements of game logic. 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 with the *Math.random()* method [1]. If used incorrectly, these PRNGs expose web applications to significant security risks. These generators lack the cryptographic robustness required

to ensure unpredictable outputs, making them susceptible to prediction and exploitation.

Our project focuses on analyzing and exploiting these weaknesses in PRNGs within JavaScript. Specifically, we utilize the Z3 theorem prover to reverse-engineer and predict past and future PRNG outputs by determining the XorShift128+ state from observed data [2]. This approach reveals the risks in relying on insecure PRNGs for sensitive or secure applications.

In addition to PRNG vulnerabilities, we examine HTTP/2's "single-packet attack", where the concurrent processing of multiple requests in a single TCP packet allows for race condition exploits. This attack method effectively disrupts the expected sequence of request handling, allowing an attacker to execute requests without interruption from other clients. By leveraging these concurrent execution vulnerabilities, our research demonstrates a practical method to exploit the predictability of web randomness in a controlled environment.

To track the flow of PRNG-generated data through web applications, we employ taint analysis, using tools such as Semgrep and CodeQL. These tools enable us to monitor data propagation and identify weak points where PRNG outputs can be exploited, being exposed to the client side. This approach facilitates a holistic assessment of web randomness vulnerabilities and provides actionable insights to protect web applications against these emerging threats.

Our paper is structured as follows: Section II explores the technical background and specific vulnerabilities in JavaScript PRNGs and HTTP/2. Section III describes the threat model that we explore in this paper. Section IV outlines our experiment of using the combined strategy from Section II on our own custom web application along with our results. Section V includes Semgrep/CodeQL rules to detect vulnerabilities to our exploit. Section VI discusses the impact of our attack, and presents mitigations and recommendations to prevent

exploitation. Section VII presents the work that remains to be done related to this topic.

II. TECHNICAL BACKGROUND: VULNERABILITIES IN JAVASCRIPT PRNGS AND HTTP/2

A. 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, rendering them vulnerable to prediction and exploitation. Developers should instead use cryptographically secure PRNGs (CSPRNGs) for cryptographic and sensitive settings.

Weaknesses in XorShift128+: The XorShift128+ PRNG, employed in JavaScript's V8 engine, 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.

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 . The algorithm can be described with the following Python code [3]:

Listing 1: XorShift128+ algorithm

```
1 def xorshift(s0: int, s1: int) ->
  tuple[int, int]:
2     se_s1 = s0
3     se_s0 = s1
4     s0 = se_s0
5     se_s1 ^= (se_s1 << 23) % (1 << 64)
6     se_s1 ^= ((se_s1 % (1 << 64)) >> 17) #
      Logical right shift instead of
      Arithmetic
7     se_s1 ^= se_s0
8     se_s1 ^= ((se_s0 % (1 << 64)) >> 26)
9     s1 = se_s1
10
11     return s0, s1
```

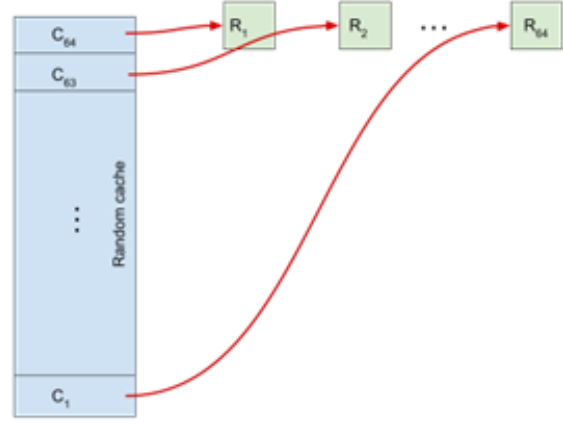


Fig. 1: Diagram of random number cache, which is first filled from bottom to top. Then when `Math.random()` is called, cache entries are outputted as shown from left to right.

To generate random numbers in the V8 JavaScript engine, a random number cache is first filled with 64 outputs of the XorShift128+ PRNG [4]. Each time a new random number is requested, an output is removed from the cache in a Last-In-First-Out (LIFO) manner. The cache is refilled when necessary. In JavaScript, `Math.random()` returns a 64-bit IEEE-format double, not an unsigned long. To do this conversion, the V8 engine will compute a mantissa from the lower 52 bits of the 64-bit s_0 . This is the portion after the decimal point of the floating-point number. Then, V8 will set the sign bit to 0 and the exponent to 1023 (0111111111), which produces a number between 1 and 2. Finally, V8 will subtract 1 from this number to produce a 64-bit double in the range 0-1 [5]. These steps are shown in the Python code below, where s_0 is the current state of the PRNG.

Listing 2: Conversion from random state to random floating point number

```
1 mantissa = s0 >> 12
2 u_long_long_64 = mantissa |
  0x3FF0000000000000
3 float_64 = struct.pack("<Q",
  u_long_long_64)
4 next_double = struct.unpack("d",
  float_64)[0]
5 next_double -= 1
```

B. HTTP/2 and Single-Packet Attacks

HTTP/2, the successor to HTTP/1.1, introduces various enhancements aimed at improving performance, such as request multiplexing, header compression, and prioritization [6]. While these features enhance user experience, they also introduce vulnerabilities that attackers can exploit. Notably, HTTP/2's multiplexing feature, which allows multiple requests to be sent in a single TCP packet, enables an attack vector known as the single-packet attack.



Fig. 2: Diagram of TCP packets used in a single-packet attack, where the last packet contains the last byte of each request.

Single-Packet Attack Methodology: The single-packet attack leverages HTTP/2 multiplexing to send multiple crafted requests that the server processes concurrently. For example, an attacker could craft a packet with a mix of incomplete and complete requests, guiding the server to process them in an order advantageous to the attacker. Usually, the attack is carried out by sending many incomplete HTTP requests spread across many TCP packets. In the last TCP packet, however, we include the last byte of each HTTP request. This allows for manipulation of resource access or data leakage, as the server’s assumptions about sequential request processing are disrupted.

Security Implications of Single-Packet Attacks: This attack vector poses significant challenges for applications that assume sequential request handling. By exploiting the concurrent processing feature of HTTP/2, attackers can potentially bypass access controls, manipulate application logic, and induce unauthorized access to resources. Applications based on HTTP/1.1’s strict request order handling are particularly vulnerable to these attacks, highlighting the need for secure request handling mechanisms in HTTP/2 environments. An attacker can also force their requests to be processed atomically, without interruption from other clients, which is especially important in a PRNG cracking context.

Combining PRNG and HTTP/2 Vulnerabilities: The combination of predictable PRNG states with HTTP/2’s concurrency capabilities creates an even more potent attack strategy. By using a single packet attack, attackers can atomically gain information about the PRNG and predict its next state and have a guarantee that no other client requested the PRNG in the middle. This can cause attackers to gain insight into web randomness that may be used for session identifiers, tokens, or other security-critical values.

C. Related Work

Various studies have examined the vulnerabilities inherent in PRNGs and their implications for security. Attackers have successfully used insecure PRNGs to break many supposedly secure protocols. Some examples include breaking Netscape’s early SSL protocol, recovering an ECDSA private key used to sign PlayStation 3 software, and attacking insecurely-generated RSA private keys [7] [8] [9]. There has also been discussion about backdoors, or intentional insecurities placed into a PRNG, placed in the NSA’s elliptic-curve based PRNG algorithm, DUAL_EC_DRBG [10].

Researchers have especially focused on the weakness of commonly used PRNGs in JavaScript, such as XorShift128+, and their potential for exploitation in web applications. XorShift128+ is of particular interest in implementation as it is favored for its speed but is not cryptographically secure. To instead use a cryptographically secure PRNG would come at a performance cost some developers do not want or are unaware they need [11].

Recent efforts have explored the use of symbolic execution to reverse-engineer PRNGs, enabling the recovery of internal states based on observable output. Notably, Douglas Goddard’s research demonstrates how JavaScript’s implementation of XorShift128+ can be exploited to predict future values generated by *Math.random()* [5]. By leveraging SMT solvers such as Z3, Goddard constructed symbolic representations of XorShift128+ to recover the lower 53 bits of its internal state from the outputted floating-point values. His work highlights how browser-specific implementations of *Math.random()* facilitate reverse-engineering through subtle variations in the conversion of 64-bit integers to doubles.

Additionally, James Kettle’s research on web race conditions further underscores the risks associated with deterministic systems. Kettle introduces the concept of single-packet attacks [12]. His approach develops a HTTP/2 single-packet attack to bypass the challenges posed by network jitter in exploiting race conditions. This technique not only broadens the understanding of race condition exploitation but also demonstrates how precise timing attacks can manipulate states in critical web applications. The implications extend beyond traditional race conditions, as a single-packet payload guarantees atomicity in its handling, which greatly enables PRNG exploitation.

Exploiting PRNG-related vulnerabilities is greatly aided by taint analysis tools. These tools can be used to demonstrate data flow across a program, from a source to a sink. Taint analysis provides a systematic approach to tracking how untrusted inputs influence the execution of programs, and how programs may reveal unwanted information in their outputs. Therefore, taint analysis is an invaluable tool in identifying vulnerabilities related to PRNGs. Tools such as Semgrep, CodeQL, and Jalangi enable developers to detect insecure data flows and potential exploits pathways, such as those introduced by predictable randomness or race conditions, using both static and dynamic analysis. By integrating symbolic execution, taint propagation tracking, and runtime monitoring, these tools enhance the ability to uncover subtle vulnerabilities and information disclosure.

III. THREAT MODEL

We will design an exploit for certain applications that use web randomness in a specific manner. Applications are vulnerable to our exploit if they meet the following three conditions

- 1) They are deployed as a HTTP/2 application



Fig. 3: Web application that we built to test our attack, displaying the result when a user guesses a number incorrectly.

- 2) They use a JavaScript backend which calls *Math.random()*, and passes this data to a user's client
- 3) They are open-source or source-available

With these requirements, an attacker may be able to analyze random data they receive, reverse-engineer the random state of the application, and predict future random states of the application. This has many potential security risks. For example, if the application uses *Math.random()* to generate unique IDs, an attacker can predict future IDs assigned to different users and compromise their confidential information or impersonate these users.

IV. THE EXPLOIT AND RESULTS

We created an example application in TypeScript that generates a random number (using JavaScript's *Math.random()*) and allows a user to guess the number: This application is written with Deno, a JavaScript runtime that uses the V8 engine [13].

This allows us to easily and ethically test the different strategies discussed earlier. The code used to generate the random number is below:

Listing 3: Code to generate random numbers in JavaScript

```
1 const generateRandomNumber = () =>
  Math.floor(Math.random() *
    10000000000) + 1;
```

This code generates random numbers in the range of 1-1,000,000,000, inclusive. These numbers are displayed to the user if they incorrectly guess the next random number. Therefore, users can see partial outputs of *Math.random()* by viewing the “correct” numbers.

A. 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, one creates a solver

object that contains a “state”. Then, one inputs “constraints” on these states that must be satisfied in the solution. Then, one can check if the solver is satisfiable (SAT) or unsatisfiable (UNSAT). If satisfiable, one can also determine a solution that satisfies all the constraints. 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, we 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, we 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 from *Math.random()*, described in subsection II-A. Then, to predict the next state of XorShift128+, we check if the solver is SAT and determine the solution.

Since *Math.random()* outputs numbers from the random cache in a LIFO manner, we must reverse our observed sequence of random numbers before inputting the numbers into our Z3 solver. Additionally, to predict the next number output from *Math.random()*, we are really predicting an XorShift128+ state that generates the numbers we have already observed. Therefore, our goal in the Z3 solver is the original XorShift128+ state that existed before the sequence of numbers we observed. Finally, since the random cache size is only 64, we observe that it becomes much more difficult to predict random numbers once we reach a cache refill boundary.

Sometimes, we find it necessary to predict random numbers more than 64 calls to *Math.random()* in the future. In this case, it becomes necessary to work around the random cache's LIFO behavior. By receiving at least 65 samples from *Math.random()*, it is guaranteed that in between two of the samples, there has been a cache refill event. This means that there exists an index i where between the random samples s_i and s_{i+1} , there was a cache refill. If we have samples s_1, \dots, s_{65} from *Math.random()*, let i be the index of a cache refill event. Suppose we are testing an index j and determining if $j = i$. Then, using Z3, we can input the prefix: s_1, \dots, s_j and attempt to predict the next random sample. Let our guess of the next random state be r_{j+1}' , which corresponds to the guess of the next random sample: s_{i+1}' . Recall that r_{j+1}' is the XorShift128+ state that comes before r_j (the associated random state to s_j), since the cache is read out LIFO. If $s_{j+1}' = s_{j+1}$, then we know that $s_j \rightarrow s_{j+1}$ does not contain a cache refill boundary, since the samples $s_{j+1}, s_j, s_{j-1}, \dots, s_1$ are indeed consecutively returned from XorShift128+. This means $i > j$. However, if $s_{i+1}' \neq s_{i+1}$, then we know that the cache refill event occurs either at j or before j , since the samples $s_{j+1}, s_j, s_{j-1}, \dots, s_1$ are not consecutive XorShift128+ outputs. This means $i \leq j$. With this information, we can perform a binary search to determine

i .

Once i is known, we know that r_{i+1}' (our guess for the next random state) is the random state of XorShift128+ directly before r_i . Let C^0 be the contents of the random cache when $\text{Math.random}()$ returns r_i . C^{-1} is the contents of the random cache before it is refilled to have contents C^0 . When the random cache is refilled after it has contents C^0 , then it will have contents C^1 . r_i is at the bottom of C^0 and r_{i+1}' is at the top of C^{-1} . All the random states r_{i+1}, \dots, r_{65} associated with samples s_{i+1}, \dots, s_{65} are members of C^1 . Since we have determined r_i , we can determine how C^0 is filled. This also lets us determine how C^1 is filled subsequently. We then simulate the random cache, first filling an array with the same states as C^1 . Then, we pop states from C^1 until we pop the state r_{65} associated with the sample s_{65} . At this point, we can determine future samples of $\text{Math.random}()$ up until s_{i+64} (since C^1 has 64 values). We also need to refill the cache if needed, using an initial state of r_{i+1} (the top of C^1). This allows us to predict future samples of $\text{Math.random}()$ indefinitely forward in the future.

We created an API to crack random numbers, where an attacker only has to input a sequence of random numbers they observe, and then they can predict the next one they will see. This API is heavily based on *v8-randomness-predictor*, a prior implementation [14]. There are two prediction modes. One is when an attacker observes raw values from $\text{Math.random}()$, which is fairly straightforwardly transformed into Z3 constraints. The second mode is when an attacker observes values of the form $\text{Math.floor}(\text{Math.random()} * N)$, or random integers from 0 to $N-1$, inclusive. In this case, the constraints inputted to Z3 need to be inequalities instead of equalities, as we only know a range of possible values of $\text{Math.random}()$ that generated the numbers we observe.

B. Sending single-packet attacks

To obtain consecutive $\text{Math.random}()$ outputs from our server, we perform a single-packet attack using the *h2spacex* library [15]. Our exploit first generates a set of n HTTP/2 streams, each with a HTTP header data frame and a HTTP body data frame, using the *h2spacex* API. Then, we send out all of the header frames and wait for the frames to arrive on the server. Finally, we concatenate all of the body data frames into a single TCP packet and send the final packet, completing the n requests. We then receive n responses for our n streams.

In testing, we found that the maximum n for our exploit was bounded by network constraints. On an instance of our test server hosted behind Cloudflare, we found that we could send a maximum of 110 concurrent HTTP streams before the stream became corrupt and the Cloudflare layer issued a GoAway frame. Note that this is 10 more than the 100 concurrent streams that the Cloudflare layer claims to support in the response header (bolded below). Nevertheless, our PRNG exploit requires 65 requests at maximum to succeed, well below the limit of 110.

Listing 4: GoAway frames sent by Cloudflare when exceeding the maximum number of concurrent HTTP/2 streams

```

1 [<H2Setting id=Max concurrent streams
   value=100 |>, <H2Setting id=Initial
   window size value=65536 |>,
   <H2Setting id=Max frame size
   value=16777215 |>]
2 # Error in sending bytes: [SSL:
   BAD_LENGTH] bad length (_ssl.c:2417)
3 ...
4 Frame Type: <H2GoAwayFrame'> / Type ID: 7

```

In addition, after modifying our test server to return a globally unique sequence number for each response, we found that single-packet requests can still be completed on a remote server in a different order than originally issued. This means that HTTP/2 stream ids *do not* necessarily correspond to the actual sequence of events server-side. Thus, in order to execute this attack on a real web server, we may need to find some sort of sequence discriminator (e.g. a timestamp) sent back by the server.

C. Unified Proof of Concept Exploit

We developed a combined exploit for our example web application that successfully predicts random numbers. First, a single-packet payload is sent to the server and the response is observed. Our payload contains many incorrect guesses of the random number, so the response contains many “correct” numbers. Due to the nature of single-packet attacks, the server atomically processed our requests and therefore, the response is guaranteed to represent consecutive calls to $\text{Math.random}()$. We then parse the response data and feed it to a script that predicts the next random value of the web application. This guess is then sent to the application and we can successfully guess the number.

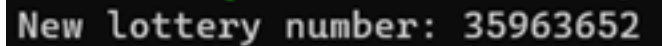


Fig. 4: The result of running our attack script, which outputs the next random number for the example application

During testing, we observed that the response from the single-packet attack is out of order from our request order. Preserving this ordering is crucial when running the random cracking exploit. Unfortunately, as described before, the connection-provided “stream ID” values did not correspond to the request order. Therefore, we had to provide a globally unique sequence number in each server response.

Future Prediction: We also simulated a testing environment where other clients are also continuously requesting the server’s random numbers during our exploit. We call this the spamming-environment model. This was done by setting up a script that frequently and periodically requests the server. We observed that our attack script still successfully predicted a new random number from $\text{Math.random}()$ in this environment. However, since there is a delay between the



Fig. 5: Our application displaying the result when guessing the next random number (see Figure 4) correctly

attacker’s first single-packet payload and then determining the PRNG state, multiple requests from other clients occur during this period. Therefore, in this environment, our attacker successfully predicts the next PRNG state of the server but cannot send it to the server as a guess fast enough. To counteract this, instead of sending the server the next PRNG state, we send the server a PRNG state that *Math.random()* will return in the future. Then, we can send many single-packet payloads with each one containing this guess. The goal is for the server to “catch up” to this future PRNG state, at which point we will have successfully guessed a future random number. The ideal number of samples to skip in the future is unclear and depends on many factors, but we find that by skipping forward 200 samples, our exploit has a high success rate. Since a single-packet attack has a limit of 110 requests, we need to send two single-packet payloads. The result of running in this environment is shown below:

Listing 5: Running our spamming-environment exploit

```

1 Setting up...
2 Initializing spamming script...
3 Running single packet attack...
4 Computing future lottery number...
5 Future lottery number: 968692383
6 Single packet attack to test future
  lottery number...
7 Found: b'{"seq":86783,"correct":true}'
8 Cleaning up...
```

Overall, we find that in both the simple and the spamming-environment models, we can successfully generate a prediction of a future random number chosen by the server. However, in the spamming-environment model, we have to guess up to 200 times until the server “catches up” to us.

To find real-world examples of applications corresponding to our exploit model, we leveraged two popular static code scanning tools, Semgrep and CodeQL [16] [17]. Both of these tools perform a type of static code analysis termed taint analysis, which tracks if a certain value, or taint, can flow from a source to a sink in a program [18]. We opted out of using dynamic taint analysis due to the high complexity to set up. For our exploit model, our source is the output from any call to *Math.random()*, and our sink is any place where that value may be transmitted to an end-user, especially the JS server-side conventions of *new Response(...)* and *res.send(...)*.

A. Semgrep

We utilized Semgrep, a static analysis tool, and experimented with different taint tracking configurations to track *Math.random()* calls to various different sinks. While performing some initial testing with our example application, we discovered that the free public version of Semgrep (Semgrep OSS) is unable to perform cross-function or cross-file analysis [19]. This means that we have to manually trace the source *Math.random()* calls across different functions, so we cannot fully generalize for various projects. Thus, Semgrep is not a great option if we want to scan through a large repository of projects. To utilize the full potential of Semgrep and perform a full taint analysis, we would need Semgrep Code, the paid version of the software.

Despite the limitation, we were still able to develop a sample of tracing the flow in our example application by setting Semgrep rules tailored to our code:

Listing 6: Semgrep rule to trace the flow of *Math.random()* in our sample application

```

1 rules:
2   - id: find_random
3     pattern-sources:
4       - pattern: "Math.random(...)"
5     pattern-sinks:
6       - pattern: "()" => ..."
7     mode: taint
8   - id: find_calls
9     pattern-sources:
10      - pattern:
11        "generateRandomNumber(...)"
12      pattern-sinks:
13        - pattern: "return new
14          Response(...)"
15      mode: taint
```

In this example, *find_random* taints an output of *Math.random()* and marks its flow through any anonymous function. In our case, we know the function we need to track is *generateRandomNumber()*, so we set up another rule *find_calls* which taints the output of this function and marks the flow through where it is returned to the user: *return new Response({tainted_output})*.

We also developed a generalized set of rules to demonstrate examples of different ways the value might be displayed to the user, ranging from various JS-server side conventions to return or console log statements.

Listing 7: General Semgrep rules to trace the flow of *Math.random()*

```

1 rules:
2   - id: find_random
3     pattern-sources:
4       - pattern: "Math.random(...)"
5     pattern-sinks:
6       - pattern: |
7         console.log($X)
8       - pattern: |
9         alert($X)
10      - pattern: |
11        res.send($X)
12      - pattern: |
13        new Response($X, ...)
14      - pattern: |
15        () => $X
16      - pattern: |
17        return $X
18      - pattern: |
19        throw $X
20    mode: taint

```

B. CodeQL

Since we were unable to perform cross-function analysis with Semgrep, we next investigated CodeQL, another static analysis tool provided by Microsoft. Notably, unlike Semgrep OSS, CodeQL provides cross-function and cross-file taint tracking out of the box for free. This requires implementing an instance of a *taint tracking configuration* class, and then executing an SQL-like query:

Listing 8: CodeQL taint tracking configuration class

```

1 from MathRandomTaintConfig cfg, Node
   source, Node sink
2 where cfg.hasFlow(source, sink)
3 select sink, "Math.random exposed to
   user from $@.", source, "here"

```

We thus implemented a custom taint tracking configuration to track values from calls to *Math.random()* to the builtin *Http::ResponseSendArgument* sink. In particular, since *Math.random()* produces a primitive number, we had to override the *isAdditionalTaintStep* in order to track taint across all numerical expressions, which is not part of CodeQL’s default taint-tracking behavior:

Listing 9: Overwriting *isAdditionalTaintStep* in CodeQL

```

1 override predicate
   isAdditionalTaintStep(Node pred, Node
   succ) {
2   succ.asExpr().(Expr).getAChildExpr() =
3   pred.asExpr()
4 }

```

Running this taint tracking configuration against our sample application reveals that our application does in fact expose a *Math.random()* value to the end-user. We then ran this taint tracking configuration against the top 100 Javascript repositories on Github using CodeQL’s multi-repository variant analysis, but found no major vulnerabilities; most usages of *Math.random()* were related to unit and integration testing scenarios [20]. One notable exception was a usage of *Math.random()* as a sentinel value in a polyfill of the *queueMicrotask* JS builtin used by the popular HTTP request library Axios [21] [22]:

Listing 10: *Math.random()* output exposed to the user in the Axios library

```

1 return postMessageSupported ? ((token,
   callbacks) => { /** ...
   */})('axios@${Math.random()}', []) :
   (cb) => setTimeout(cb);

```

Given that all modern browsers support *queueMicrotask*, and that exploiting this vulnerability will likely already require arbitrary code execution, we believe this usage of *Math.random()* is unlikely to cause issues.

VI. DISCUSSION AND MITIGATIONS

Our threat model and exploit present a significant risk to applications that need to send users “secret” data that other users must not know. While the threat model is somewhat restrictive, many applications may inappropriately use *Math.random()*, allowing for these secrets to be compromised. These secrets could include UUIDs, CSRF tokens, room codes, and more. If an attacker knows the state of XorShift128+ and the random cache, they can step forward and backward an arbitrary number of steps, which means that all past and future random outputs will be known to the attacker.

For applications that may be vulnerable according to our threat model, developers need to first assess the risks associated with a potential exploit. If random numbers are generated with *Math.random()* in the application backend, and are used for sensitive purposes (e.g. generating universal unique identifiers, or UUIDs, that can be used to access sensitive personal information), attackers may be able to generate a sequence of random numbers and learn the random data of other users of the application. This is something that should be mitigated. On the other hand, if random numbers from *Math.random()* are used to generate non-sensitive information (e.g. room join codes for an online game), a developer may

choose to accept the risk of an attacker learning this random information.

To mitigate or eliminate the risk of exploitation in the former scenario described above, we offer two recommendations: using a cryptographically secure pseudorandom number generator (CSPRNG) and/or implementing additional authentication/authorization. These recommendations are not mutually exclusive, but there are reasons for developers to use one solution or the other.

Using a well-known CSPRNG will eliminate the risk of an attacker executing our exploit. A CSPRNG will produce random numbers with important cryptographic properties. First, by knowing any number of outputs of a CSPRNG, an attacker should be no more likely to predict the next bit outputted by the CSPRNG (or, equivalently, learn the CSPRNG’s internal state). Additionally, by knowing a CSPRNG’s internal state, an attacker should not be able to learn the CSPRNG’s past outputs. With these requirements, a CSPRNG models a pseudo-random generator (PRG), a theoretical device in cryptography that produces an output that is indistinguishable from random. An example of a CSPRNG algorithm available for JavaScript developers is the `crypto.getRandomValues()` method [23]. However, CSPRNG algorithms tend to be computationally expensive and therefore may incur undesired overhead.

Developers can also implement authentication/authorization to mitigate some risks of our exploit. For example, if randomness is used to generate UUIDs, users can still be required to authenticate themselves using a password before accessing sensitive information. This way, even if an attacker knows another user’s UUID, they cannot access their sensitive information. However, this mitigation does not remove all risks of exploitation. If only this mitigation is present, attackers are still able to learn other users’ random information. For example, if `Math.random()` is used to generate authentication tokens such as CSRF tokens, an attacker can still predict the values of such tokens, and include this information in a larger CSRF exploit that leverages an already-authenticated victim.

VII. FUTURE WORK

We currently rely on the server returning an indication of the order the responses should be in (via sequence number). This is not a guarantee for applications in the wild and we need another way to decipher the ordering of these responses. A potential solution is to brute-force the ordering of the responses. We have observed that it is extremely unlikely for a different ordering of the same outputs of `Math.random()` to be valid. A valid ordering is one where there exists a 2^{*64} -bit state s_0, s_1 for XorShift128+ that can generate that sequence of random numbers from `Math.random()`. Therefore, with enough samples of random numbers (around 8-10), it is likely that we only find one valid ordering of outputs, which is the true ordering of outputs sent from the server. However, brute forcing ordering of a sequence is an extremely computationally intensive problem. We are only able to brute force the ordering of up to 8 samples in reasonable time

for our exploit, since the time complexity to determine the ordering of n samples is $O(n!)$. Another possibility is that HTTP/2 responses could include data that can be used to reconstruct the order of responses. However, we have not found such data yet and we need to do further research into the protocol.

While single-packet attacks have been studied with HTTP/2 as a way to enable race conditions to execute more reliably, it is still possible to divide HTTP request fragments on the transport layer in HTTP/1.1. Therefore, we may be able to expand our attack to HTTP/1.1 applications and there also may be better ordering guarantees with this protocol, circumventing our prior ordering issues [12].

Another thing to try would be to use dynamic taint analysis to track the flow of `Math.random()` dynamically instead of statically. Unlike static analysis, dynamic taint analysis operates at runtime, which allows for the identification of vulnerabilities in code paths that are only triggered under specific conditions. For instance, certain branches in the program logic might execute only with specific user inputs, configurations, or states that static analysis might miss. By instrumenting the JavaScript runtime or integrating with tools like Jalangi, we can monitor the actual execution of the program and track how `Math.random()` outputs propagate through the application in real-time [24].

Finally, we can expand our taint analysis to outside the top 100 open-source repositories on Github to include more of the less tested projects that are likely to be vulnerable to our attack. Most of the applications we tested were libraries rather than user-facing applications; we suspect that `Math.random()` bugs are much more likely to occur in application code rather than library code.

REFERENCES

- [1] v8, “random-number-generator.h,” <https://github.com/v8/v8/blob/085fed0fb5c3b0136827b5d7c190b4bd1c23a23e/src/base/utils/random-number-generator.h>, 2024, accessed: Nov. 12 2024.
- [2] Microsoft, “Github - z3prover/z3: The z3 theorem prover,” <https://github.com/Z3Prover/z3>, 2024, accessed: Nov. 11 2024.
- [3] G. Marsaglia, “Xorshift RNGs,” *Journal of Statistical Software*, vol. 8, no. 14, 2003.
- [4] Google, “math-random.h,” <https://chromium.googlesource.com/v8/v8/+6d706ae3a0153cf0272760132b775ae06ef13b1a/src/math-random.h>, 2024, accessed: Nov. 11 2024.
- [5] D. Goddard, “Hacking the javascript lottery,” <https://blog.securityevaluators.com/hacking-the-javascript-lottery-80cc437e3b7f>, 2024, accessed: Nov. 12 2024.
- [6] CloudFlare, “Http/2 vs. http/1.1,” <https://www.cloudflare.com/learning/performance/http2-vs-http1.1/>, 2024, accessed: Nov. 11 2024.
- [7] I. Goldberg and D. Wagner, “Randomness and the netscape browser,” <https://people.eecs.berkeley.edu/~daw/papers/ddj-netscape.html>, 1996, accessed: Dec. 6 2024.
- [8] M. Bendel, “Hackers describe ps3 security as epic fail, gain unrestricted access,” <https://www.exophase.com/20540/hackers-describe-ps3-security-as-epic-fail-gain-unrestricted-access/>, 2010, accessed: Dec. 6 2024.
- [9] A. Lenstra, J. Hughes, M. Augier, J. Bos, T. Kleinjung, and C. Wachter, “Ron was wrong, whit is right,” vol. 2012, article 64, 01 2012.
- [10] J. Ball, J. Borger, and G. Greenwald, “Revealed: how us and uk spy agencies defeat internet privacy and security,” <https://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security>, 2010, accessed: Dec. 6 2013.

- [11] Y. Guo, “There’s `math.random()`, and then there’s `math.random()`,” <https://v8.dev/blog/math-random>, 2024, accessed: Dec. 6 2024.
- [12] J. Kettle, “Smashing the state machine: the true potential of web race conditions,” <https://portswigger.net/research/smashing-the-state-machine#single-packet-attack>, 2024, accessed: Nov. 11 2024.
- [13] Deno, “Deno, the next-generation javascript runtime,” <https://deno.com/>, 2024, accessed: Nov. 11 2024.
- [14] PwnFunction, “v8-randomness-predictor,” <https://github.com/PwnFunction/v8-randomness-predictor>, 2024, accessed: Nov. 12 2024.
- [15] nxenon, “h2space,” <https://github.com/nxenon/h2space>, 2024, accessed: Nov. 12 2024.
- [16] Semgrep, “semgrep.dev,” <https://semgrep.dev/>, 2024, accessed: Dec. 6 2024.
- [17] GitHub, “Codeql,” <https://codeql.github.com/>, 2024, accessed: Dec. 6 2024.
- [18] Semgrep, “Taint analysis,” semgrep.dev/docs/writing-rules/data-flow/taint-mode, 2024, accessed: Dec. 6 2024.
- [19] —, “Semgrep faq,” <https://semgrep.dev/docs/faq>, 2024, accessed: Dec. 6 2024.
- [20] GitHub, “Running codeql queries at scale with multi-repository variant analysis,” <https://docs.github.com/en/code-security/codeql-for-vs-code/getting-started-with-codeql-for-vs-code/running-codeql-queries-at-scale-with-multi-repository-variant-analysis>, 2024, accessed: Dec. 6 2024.
- [21] —, “Axios/axios: Promise based http client for the browser and node.js,” <https://github.com/axios/axios>, 2024, accessed: Dec. 6 2024.
- [22] MDN, “Window: `QueueMicrotask()` method - web apis: Mdn,” <https://developer.mozilla.org/en-US/docs/Web/API/Window/queueMicrotask>, 2024, accessed: Dec. 6 2024.
- [23] —, “Crypto: `getRandomValues()` method - web apis — mdn,” <https://developer.mozilla.org/en-US/docs/Web/API/Crypto/getRandomValues>, 2024, accessed: Nov. 11 2024.
- [24] Samsung, “Samsung/jalangi2,” <https://github.com/Samsung/jalangi2>, 2024, accessed: Nov. 11 2024.