Security Model for ECMAScript Runtimes

Abdul Malik
Computer Science and Engineering
JIIT, Noida

Ujjwal Sharma
Computer Science and Engineering
JIIT, Noida

Darsh Mecwan
Computer Science and Engineering
JIIT, Noida

***Abstract*—JavaScript has become a go to language for web application and rightly so. With the advent of technologies like NodeJS one can move server side operation on JavaScript as well, making it highly efficient and reliable as a language. It is one of the core technologies of the world wide web. However, this expansion of the JavaScript ecosystem has been made possible by the virtue of a huge database of modules and dependencies that are maintained by individuals or loosely held organisations. These modules provide functionality to the bare bones core model and principles on which JS works. This expansion of the JavaScript ecosystem comes with a huge cost. The fact that so many applications use these third party modules is testimony enough that these modules are important but there is no one system in place that checks all these modules for security breaches or vulnerabilities. If the third party code is not verified and vetted for security, by extension the application code, however robust loses it defence against possible attacks. This problem is not something new and has been known to developers for quite sometime now.Even though JavaScript makes use of JIT (just in time) parsing which is faster than most alternatives available to us, it is not just fast enough for us to implement a system of checks and balances without incurring a substantial overhead. To tackle this, we started exploring the idea of Ahead of Time parsing and how we could implement it. For this we turn towards Rust as an efficient technology tool to help us translate the available JS parser to Web Assembly. Up until now security concern was seen as a trade off to achieve the desired functionality in required time but with the recent rise of light and fast Web Assembly languages, there is hope that these can be used to complement ECMAScript and rid us of the issues associated with JavaScript implementations of resources like parsers and analysers.**

***Index Terms*—parsing, Ahead of time, ECMAScript, NodeJS, Web Assembly language**

I. Introduction

JavaScript often abbreviated as JS, is a high-level, interpreted scripting language that conforms to the ECMAScript specification. JavaScript has curly-bracket syntax, dynamic typing, prototype-based object-orientation, and first-class functions. JavaScript is one of the core technologies of the world wide web. It enables interactive pages and is an essential part of web applications. In enabling various functions JavaScript relies heavily on JavaScript Resources. The direct client side inclusion of these cross origin resources in Web Applications has become an excessively pervasive practice to consume third-party services and to utilize externally provided code libraries. Even though this inclusion increases the functionality manifold, there are several downsides to it.

Most of the issues arise as a result of the access that is granted to these third party services/libraries to access the resources essential for their functioning. The problem is that such a code runs in the same context and with the same privileges as the first party code. This gives rise to major security concerns. Vulnerabilities in third-party libraries is a growing concern for software developers, not only because it poses risks to the software client itself, but to the entire ecosystem[1][2]. For starters, all potential security problems in this third party code now directly affect the site that included these. Secondly, these third party libraries/services can be used as gateways to mount possible attacks on the sites that use these. Currently, npm has over 800,000 published packages, with each package having, on average, a total of 90 direct and indirect dependencies. As the number of packages and the frequency of updates increase in the ecosystem, it becomes impractical for the community to review all package updates on npm. This year alone, there have been approximately 4,900 updates per week (29 per hour), making it unrealistic to assume that the community can manually review all of them.[3] This problem is not something new and has been known to developers for quite sometime now.

However another problem lies in the solution to this problem. JavaScript does not provide any kind of privilege separation between code loaded from different packages. That is, any third-party package has the full privileges of the entire application. This situation is compounded by the fact that many npm packages run outside of a browser, in particular on the NodeJS platform, which does not provide any kind of sandbox[4]. All the methods proposed up till today to solve this issue were methods involving either a very big computational or time overhead or methods that restricted the functionality of the applications. This problem was owed partly to the fact that it was logistically very problematic to scan each and every module being uploaded and all the updates there after and keep a track of any vulnerability therein and partly because the nature of web browsers and the way they function makes it a difficult task to perform and achieve a real time security scan at run-time. This is owed to the fact that the parsers that convert the user codes from ECMAScript (JavaScript) for syntactic analysis have their own limitations in form of performance and functionality. These limitations are majorly credited to the language in which these parsers are often written. The fact remains that the programming language plays a major role in the way a program functions as well as it heavily affects the performance parameters of a program and hence in a way restricts the program from utilising its full potential. Upon analysis we found this to be the case with our problem statement. We as a group have been associated

with applications that make use of NodeJS and realise the importance of modules and the diversity they bring to this technology, however, we have also realised that in its present state NodeJS and all technologies using third party JavaScript are highly vulnerable. Up until now this security concern was seen as a trade off to achieve the desired functionality but with the recent rise of light and fast Web Assembly languages, there is hope that these can be used to complement ECMAScript and rid us of the issues associated with JavaScript implementations of resources like parsers and analysers. This study is an attempt to understand the working of Web Browsers and the technologies that utilize these web browsers along with third party modules to enhance functionality of applications. Considering that the security of our resources is a legitimate concern, through this study, we aim to identify the ways in which we can mitigate the concerns by making the smallest trade off possible. For the purpose of intensive study and circumstantial understanding, we have restricted ourselves to Rust for coding purposes. The reason for doing so has been illustrated further.

II. Idea behind Securing Web Applications

Web applications that are developed with hard time constraints are often deployed with critical software bugs, which are invisible at the time of deployment either due to bad development practices or intentionally, making them vulnerable to attacks. Perhaps unsurprisingly, npm's openness comes with security risks, as evidenced by several recent incidents that broke or attacked software running on millions of computers[4]. To understand the purpose and use of our research, it is imperative to understand the classification and knowledge of the typical software bugs that lead to security vulnerabilities. For the purpose of which we refer to previous work analyzing some major security patches of widely used web applications[5]. To make it a qualitative research, we restricted the vulnerabilities listed and tested to the most commonly listed ones on CVE and the analysis results were compared against other field studies on general software faults (i.e., faults not specifically related to security). Furthermore, a detailed analysis of the code of the patches shows that web application vulnerabilities result from software bugs affecting a restricted collection of statements or overuse of a system resource or unrestricted access to a particular system resource. This finding in itself is very important as it forms the bedrock of our solution system. The existing work in identification and patching[5] uses a brute force approach and works in devising a patch after identifying a fault in the implementation and solving it for a future version and this cycle continues.

Another interesting phenomena that hinders the developers from putting security mechanism in place is the fact that in practice programs written with security optimisation should run as fast as their un-optimised counterparts. They should, but they don't. A major reason for this is the level of optimisation applied to these two classifications create a significant overhead. Further, the language that is used for programming also significantly affects the time when developing a software or code[6]. The fact is that JavaScript has come to be known as a very fast and indispensable language when it comes to web development but the fact remains that in certain conditions implementations like parsing and analysis, assembly languages still beat JavaScript and the recent rise of strong, robust and unambiguous assembly languages like Rust and Go has reinforced our findings that assembly languages are the way to go to achieve best case time complexities when working in a service dependent ecosystem.

As a rule general, it has become virtually impossible in recent years to develop any significant software systems without relying at least to some degree on available component ecosystems, such as the npm ecosystem previously mentioned. Opportunistic reuse is common despite the risks that components developed by unknown developers, using unknown methodologies, may contain unknown and possibly harmful safety related characteristics[8]. As JavaScript is becoming more and more popular, including the server-side NodeJS platform, which advocates a single-threaded, event-based execution model that uses asynchronous I/O calls. In NodeJS, the main thread of execution runs an event loop, called the main loop that handles events triggered by network requests, I/O operations, timers, etc. A slow computation, e.g., matching a string against a regular expression, slows down all other incoming requests[1][9] and these platforms have become an inseparable part of the web ecosystem. Like most ecosystems, the web comprises of complex web applications that mash up scripts from different origins inside a single execution context in a user's browser. This execution scheme opens the door for attackers, too. Vulnerability studies consistently rank Cross Site Scripting(XSS) highest in the list of the most prevalent type of attacks on web applications. Attackers use XSS to gain access to confidential user information[7]. The approaches used up until now to prevent misappropriation of sensitive data introduce significant run-time overheads that make execution of JS code at least two to three times slower. Due to this, shortcoming the industry will never adopt this information flow approach without a substantial reduction in this overhead[7]. The various approaches being worked on are those of distributing the tracking workload across all page visitors by probabilistically switching between two JavaScript execution modes. Though independent from the idea of security this concept is important as it gives us a clear picture as to why we need to take in account the time and resource overhead when developing a feasible solution.

III. Approaches presently being used to secure Applications

A. Sandboxing

One of the popular security mechanism being used in certain web applications these days employs the concept of sandboxing for executing untrusted code and enforce established security policies. Although sandboxing techniques have individual strengths, they also have limitations that reduce the scope of their applicability[10]. Different sandboxing mechanisms have specific design trade-offs and developing a robust sandboxing

facility that combines the strengths of wide variety of design alternatives has been a matter of constant study. Even though sandboxing is a widely used security mechanism that works for ensuring safety of applications, this technique has a major flaw. The fact that third party modules when integrated with core modules, act on behalf of these core modules and even through the sandbox gain access to all the resources that the core module has access to. This design serves as an attack window for exploitation and this design flaw is not tackled by sandboxing methods.

B. XSS- Safe

A framework known as XSS- SAFE which is a server-side automated framework for the detection and mitigation of XSS attacks. It is designed based on the idea of injecting the features of JavaScript and introduces an idea of injecting the sanitizing routines in the source code of JavaScript to detect and mitigate the malicious injected XSS attack vector[11]. The sanitizing routine checks for known and unknown XSS attacks with minimum false positives. The idea is to match the suspicious code with an existing sample space to find a probable match. The framework is a damage control mechanism and does not provide preemptive security. It works on the detection after the XSS has been injected and started doing what it was intended do to, hence making it a passive defender when it comes to security.

C. Native Client

Another sandboxing alternative, the design, implementation and evaluation of a Client, for untrusted native code.The aim is to give browser-based applications the computational performance of native applications without compromising safety by using software fault isolation and a secure runtime to direct system interaction and side effects through interfaces managed by Native Client[12]. The client is usually a system that handles untrusted modules from any web site with comparable safety to accepted systems such as JavaScript. An untrusted module may contain arbitrary code and data. The client proposes some rules that the modules have to conform to. If the modules don't conform to these rules, they are rejected by the system.This system though enforces some rules and checks but the problem with this implementation is that it does not provide a defense in depth mechanism for modules that may conform to the conventions and yet contain malicious or dangerous code. It is just a superficial safety mechanism that may filter out code that does not conform to the guidelines setup by the native client.

D. Hybrid Feature Extraction

Since technology is booming, there is an increase in need for securing web Applications. Due to the hike in Internet Users the web services give rise to new challenges. One of major challenge is to secure against malicious attacks. This framework focuses on detecting XSS vulnerability of all flavors. First WAF status is detected based on the error code. Then filter checker is applied to determine the presence of Reflected XSS vulnerability and Payloads are generated to check for Blind XSS. The experimental results show that there lot of sites of vulnerable to XSS. Hybrid Feature extraction is a combination of Web Application Firewall Detector, Fuzz testing and filter checker[13].

E. Script Protect

Script Protect is a non-intrusive transparent protective measure to address security issues introduced by external script resources. Script Protect automatically strips third-party code from the ability to conduct unsafe string-to-code conversions. Thus, it effectively removes the root-cause of Client-Side XSS without affecting first-party code in this respective. Script Protect is realized through a lightweight JavaScript instrumentation, it does not require changes to the browser but incurs a low run-time overhead of about 6 percent [14].

F. Attribute based Access Control

Attribute-based access control (ABAC) is a promising alternative to traditional models of access control (i.e.,discretionary access control (DAC), mandatory access control (MAC), and role-based access control (RBAC)). It is an access control method where subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, environmental conditions, and a set of policies that are specified in terms of those attributes and conditions. Even though it has a promising future, ABAC research is still largely in its infancy, the list of open problems related to ABAC systems and implementations is extensive. The majority of these problems stem from the increased complexity attribute, and policy-based access control introduces for the sake of increasing the flexibility and generality of access control policies[15].

G. High performance Parsing Engines

High performance JavaScript engines are a prototype to reimagine dynamic nature of JavaScript. The main components of the developed engine, are execution manager, semantics analyzer, type information manager, persistent cache of code and execution profile, JS parser, and optimizing compiler. The overall execution process is managed by execution engine, which handles an executed application's source code, loads the available information about the application from persistent cache, and starts with most effective way of application execution, which is available according to the cached data. Possible ways are generic interpretation, optimized interpretation backed by inline caches, fast-compiled region invocation, and finally execution of fully optimized AOT-compiled region representing the whole program[16].

IV. Detection Mechanism

It has become common practice for software projects to adopt third-party libraries, allowing developers full access to functions that otherwise will take time and effort to create[17]. Owing to this, the internet is bustling with various security issues in open source software package libraries.Another malice

plaguing the ecosystems is the fact that with an increased level of automation provided by package managers, which sometimes allow updates to be installed automatically, malicious package updates are becoming a real threat in such software ecosystems[3]. Hence, it is a very tedious task to discover and fix vulnerabilities in packages. In addition, such issues may propagate to dependent packages, making them vulnerable too. Even though a lot of resources online give regular updates on emerging security loopholes and trends, the reservoir is not enough to ascertain the impact the security compromise can have on individual applications. It is essential to take into account the severity of these breaches and thus identify to which extent they can affect other packages in the packaging ecosystem in presence of dependency constraints[18]. Another important link in devising an efficient detection mechanism is identifying why these threats hereafter referred to as vulnerabilities, arise in the first place and what are the most susceptible paths through which these vulnerabilities propagate. Vulnerability essentially refers to any weakness in system's security requirement, design, coding or operation that could accidentally occur or intentionally violated and results in a security failure[19].

Once the basic outline has been created on which vulnerabilities one is targeting and its paths are all analysed, comes the role of static analysis. Static Analysis is an indispensable tool used by developers to search for vulnerabilities in the source code of web applications. However, distinct tools provide different results depending on factors such as the complexity of the code under analysis and the application scenario; thus, missing some of the vulnerabilities while reporting false positives.In recent years some of such tools have used methods like meta-programming and code rewriting[20], Heuristics[21], quality measurement tools for the registry itself[22] like npm-miner and some very innovative inventions like the MTD[moving target defence] techniques to improve the network security by continuously re-configuring the system settings[23].

Benchmarking systems like npm-miner can be very effective to assess and compare different systems or components, however, existing benchmarks have strong representative limitations, disregarding the specificity of the environment, where the tools under benchmarking will be used[8]. For this very reason assessing and comparing static analysis tools in terms of their capability to detect security vulnerabilities becomes important. The effectiveness of how secure the environment is will directly be dependent on the ability of the Static Analysis run-time in identifying the said vulnerability. In a nutshell, to ensure an effective identification mechanism, we need to outline robust parameters that we wish to tackle through the solution. The ecosystem that we are targeting, i.e. the ECMAScript ecosystem has an interesting track record when it comes to identification of security issues. The fact is that the popularity of JavaScript has lead to a large ecosystem of third-party packages available via the npm software package registry. The open nature of npm has boosted its growth, providing over 800,000 free and reusable software packages.[9] Unfortunately, This open nature also causes security risks, as evidenced by recent incidents of single packages that broke or attacked software running on millions of computers. The security risks for users of npm can be easily understood by systematically analyzing dependencies between packages, the maintainers responsible for these packages, and publicly reported security issues. Our study revealed that the vulnerabilities run into millions and when different packages are clubbed together they give rise to exponentially more issues.

V. Dependence on choice of Programming Language

In this section, we describe the reasons for changing the programming language. Language-based ecosystems (LBE), i.e., software ecosystems based on a single programming language, are very common. Examples include the npm ecosystem for JavaScript, and PyPI for Python. These environments encourage code reuse between packages, and incorporate utilities—package managers—for automatically resolving dependencies[24]. However, the same aspects that make these systems popular—ease of publishing code and importing external code—also create novel security issues, which have so far seen little study. Reusable Open Source Software (OSS) components for major programming languages are available in package repositories. Developers rely on package management tools to automate deployments, specifying which package releases satisfy the needs of their applications. However, these specifications may lead to deploying package releases that are outdated, or otherwise undesirable, because they do not include bug fixes, security fixes, or new functionality. In contrast, automatically updating to a more recent release may introduce incompatibility issues[25].

Such is the nature of these language based ecosystems that a solution at the repository level will always come with added challenges and costs. A solution needs to come from the foundation on which such ecosystems stand to nullify any costs or overheads that may incur. In theory, Programs written in powerful, higher-order languages like Scheme, ML, and Common Lisp should run as fast as their FORTRAN and C counterparts. They should, but they don't[6]. A major reason is the level of optimisation applied to these two classes of languages. Similarly, programs and modules written in JavaScript are extremely fast in theory and in practice but a faster alternative that is compatible with the ecosystem can provide us the advantage that we are looking for. With the rise of Web Assembly languages this vision is achievable and practical as these languages provide the speed of Assembly languages without compromising on the functionality of JavaScript. Web Assembly is a new type of code that can be rotated in modern web browsers. It offers new functions and has great merit in terms of performance. It is designed to be an effective compilation target for low-level source languages such as C, C ++, Rust. Web assemblies have significant implications on the Web platform. It provides a way to execute code written in multiple languages on the Web at a speed close to native,

making it possible to run client applications that could not be executed on the Web until now[18].

In this bid to utilise the power of these web assembly languages, we turned to Rust. Rust is a programming language spearheaded by Mozilla. It is a general-purpose programming language emphasizing memory safety and speed concerns.Rust guarantees memory safety in a unique way compared to other commonly used programming languages. The compiler statically analyzes the source code, tracking the "lifetime" of any heap-allocated data. When all existent pointers to a piece of data has gone out of scope (e.g. at the end of a function), then the compiler determines that the data is no longer alive and the allocated space can be freed. This memory management system prevents many classic memory errors by disallowing the programmer from accessing uninitialized or already-freed memory[26]. Rust code compiles down to a native executable. The Rust compiler uses an LLVM back-end to emit assembly, taking advantage of LLVM's extensive optimization options. In addition, there is no run-time associated with executing Rust binaries; Rust is not run by a virtual machine (e.g. Java), nor does it use garbage collection during execution[26].

Owing to these great features we decided to explore this JS alternative as a viable option to translate the existing JavaScript parser 'acorn' to Rust. We chose the Rust language, because of the following properties[27][28][29]:

- Managed memory: Rust has a concept of lifetimes and ownership, which guarantees the safety of memory without requiring a garbage collector or reference counting, at compilation time. In particular, Rust prevents use after free and double free flaws.
- No garbage collector: controlled memory use and life of objects.
- Thread-safe: this is guaranteed by the compiler.
- Efficient: the source is compiled to native code, and produced code that tends to be quite fast (similar to C).
- Zero-copy: data buffers (called slices), objects and references can point to the same locations, and the compiler uses it to avoid copying data while ensuring the memory safety.
- Easy integration with other languages, as few data conversions as possible.
- Clear marking of the safe/unsafe parts of the code: all the potentially dangerous instructions for memory safety (including calls to C functions and syscalls) must be enclosed in an unsafe block, and the rest is statically verified to be safe.
- Good/large community: it's important to use a good language, but it is better to have active and helpful users.

VI. OUR SOLUTION

Considering the fact that 'Security' is a complex domain and its implementation is dependent on a wholesome understanding of the structure in which it works. In modern web-based applications, an increasing amount of source code is generated dynamically at run-time. Web applications commonly execute dynamically generated code emitted by third-party, black-box generators, run at remote sites[30].The solution that we propose is simple yet highly effective. The way JavaScript works is that it compiles the code passed to it and creates an AST file which is then structured to be displayed or used as required. Static analysis of JavaScript applications is highly challenging due to its dynamic language constructs and event driven asynchronous executions, which also give rise to many security-related bugs. Several static analysis tools to detect such bugs exist, however, research has not yet reported much on the precision and scale-ability trade-off of these analyzers[31]. If we could devise a way in which a certain static analysis could be performed on this AST before it is rendered, we can identify possible vulnerabilities before they affect the system. For this to be feasible and implementable, we need to identify or devise a way that can nullify the time overhead that we incur at the time of static analysis. To do that, we need a better more efficient parser than what we have available at the moment. Once a better and faster parser is in place, it is then time to perform static analysis on the AST produced and detect vulnerabilities.

Further, clustering this analysis into an open source, viewable module that makes is mandatory to add parameter based access to the modules which thereafter can only access resources that they have been permitted by the user. An important concept in this routine is that of a benchmark. The most common method to assess and compare the performance of alternative tools is to run them with a set of representative test cases and compare the results. A standard process for doing this task is called a benchmark[32]. In doing this analysis, we effectively do two things, firstly, we try to identify known and unknown threats at the time of compilation and before execution. Secondly, we restrict access to system resources to ensure that a person or an application can only use the resource that it is permitted to use. This approach has not been undertaken as a project up until now and is a very unique take on how security can be implemented in a JavaScript runtime environment. The implementation, however, is heavily dependent on the use of a highly efficient programming language. Although the performance of today's JavaScript engines is sufficient for most web applications, faster and more predictable run-times are desirable for the performance-critical step of parsing. The reason for the performance discrepancy is not inadequately designed JavaScript engines, but mainly inherent to the design of the JavaScript language itself that favors ease of use over performance[33]. For the purpose of this project, we have chosen Rust as our preferred language owing to its predictable and efficient performance, small code size and a lively ecosystem. We started the development life cycle by separating and identifying the cornerstone of our implementation. In our case this corner stone is the development of a faster and better parser. For the purpose of this prototype, we have taken 'ACORN' which is the standard parser used in NodeJS and we translated it into Rust Web Assembly to add on to it the goodness that rust provides. We have named our parser 'CAP'. This step in itself is a very

important step of the process and forms the foundation of the entire model.

The second process in the implementation process is the identification or development of a suitable benchmark or process for the static analysis of the Abstract Syntax Tree. The flowchart of how the model shall function is as depicted in Fig 1.

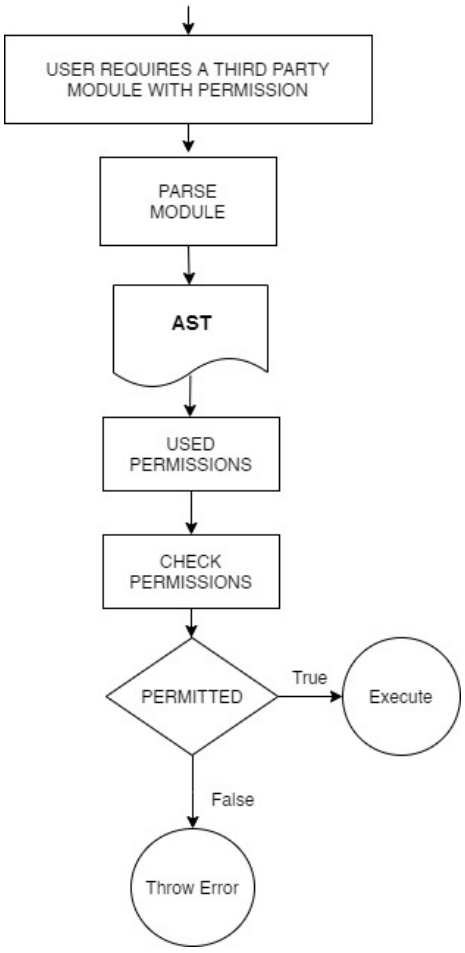

Fig. 1. Flow diagram of the model

Up until now, the NodeJS modules and libraries got implicit permissions to access all the system resources as soon as they were required into a program. The model that we propose follows a white-listing procedure where the user not only passes the name of the module with the require call but also specifies the permissions that are given to that package. In case the package tries to access something other that the white-listed resources, the system would throw an error. The way we achieve this is by analysing the AST that is produced after parsing it through our parser. Since, this method of matching access is a time consuming process we needed a enhancement to nullify the overheads that we are incurring. This process is taken care of by 'CAP' our web assembly parser for ECMAScript written in Rust. The time saved in parsing allows us to perform the static analysis on our AST file and match the permissions through an access control list or any other method that we deem fit for matching permissions, without compromising on the performance of the system. This simple approach was very effective and helped detect a major chunk of vulnerabilities. In the future, this model can be optimised to include advanced features like identification and further profiling using better benchmarks.

REFERENCES

[1] Bodin Chinthanet, Raula Gaikovina, Kula, Takashi Ishio, Akinori Ihara, Kenichi Matsumoto "On The Lag of Library Vulnerability Updates: An Investigation into the Repackage and Delivery of a Security Fix Within The npm JavaScript Ecosystem", arXiv:1907.03407v2 [cs.SE] 14 Oct 2019.

[2] Alexandre Decan, Tom Mens, Eleni Constantinou "On the impact of security vulnerabilities in the npm package dependency network", 2018 ACM/IEEE 15th International Conference on Mining Software Repositories.

[3] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, Christian Kastner "Detecting Suspicious Package Updates", 978-1-7281-1758-4/19, DOI 10.1109/ICSE-NIER.2019.00012.

[4] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, Michael Pradel "Small World with High Risks: A Study of Security Threats in the npm Ecosystem", arXiv:1902.09217v2 [cs.CR] 7 Jun 2019.

[5] Jose Fonseca, Marco Vieira "Mapping Software Faults with Web Security Vulnerabilities", 1-4244-2398-9/08/International Conference on Dependable Systems and Networks: Anchorage, Alaska, June 24-27 2008.

[6] Olin Shivers "Control-Flow Analysis of Higher-Order Languages", CMU-CS-91-145.

[7] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler and Michael Franz, "CrowdFlow: Efficient Information Flow Security", ISC 2013, LNCS 7807, pp. 321–337, 2015.

[8] Tommi Mikkonen and Antero Taivalsaari "Software Reuse in the Era of Opportunistic Design", 0740-7459/19©2019IEEE.

[9] Cristian-Alexandru Staicu and Michael Pradel "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers", ISBN 978-1-939133-04-5, 27th USENIX Security Symposium, 2018.

[10] David S. Peterson, Matt Bishop, and Raju Pandey "A Flexible Containment Mechanism for Executing Untrusted Code", 11th Annual USENIX Security Symposium Pp. 207-225.

[11] Shashank Gupta, B. B. Gupta "XSS-SAFE: A Server-Side Approach to Detect and Mitigate Cross-Site Scripting (XSS) Attacks in JavaScript Code", Arab J Sci Eng (2016) 41:897–920.

[12] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar "Native Client: A Sandbox for Portable, Untrusted x86 Native Code", 1081-6011/09 2009 IEEE.

[13] V.Subramaniyaswamy, Kalyani Gopireddy Venkata, Likhitha Naladala "Securing Web Applications from malware attacks using hybrid feature extraction", International Journal of Pure and Applied Mathematics, Academic Publishing Ltd, 2018, 119 (12), pp.13367-13385.

[14] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, Martin Johns "ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices" in ACM Asia Conference on Computer and Communications Security (AsiaCCS'19), July 9–12, 2019, Auckland, New Zealand. ACM, New York, NY, USA, 10.1145/3321705.3329841.

[15] Daniel Servos and Sylvia L. Osborn. 2017. Current research and open problems in attribute-based access control. ACM Comput. Surv. 49, 4, Article 65 (January 2017)10.1145/3007204.

[16] Ayrapetyan R.B., Gavrin E.A., Shitov A.N. "A NovelL Approach for Enhancing Performance of JavaScript Engine for Web Applications", white paper

[17] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, Akinori Ihara "Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages", 2576-3148/18/2018 IEEE International Conference on Software Maintenance and Evolution.

[18] Jin-Tae Park, Hyun-Gook Kim and Il-Young Moon "The JavaScript and Web Assembly Function Analysis to Improve Performance of Web Application", ISSN: 2005-4238 IJAST, International Journal of Advanced Science and Technology, Vol.117 (2018), pp.1-10

[19] Sajjad Rafique, Mamoona Humayun, Bushra Hamid, Ansar Abbas, Muhammad Akhtar, Kamil Iqbal "Web Application Security Vulnerabilities Detection Approaches: a Systematic Mapping Study", 978-1-4799-8676-7/15/2015 IEEE- SNPD 2015, Takamatsu, Japan.

[20] Ricardo Medel, Alexis Ferreyra, Nestor Navaro, and Emanuel Ravera "Applying Meta-Functions for Improving JavaScript Code Performance", EJS 17 (1) 15-34 (2018).

[21] Kavya Reddy Mahakala "Identifying Security Requirements using Meta-Data and Dependency Heuristics", thesis, University of Cincinnati, Fall 2018.

[22] Kyriakos C. Chatzidimitriou, Michail D. Papamichail, Themistoklis Diamantopoulos, Michail Tsapanos, and Andreas L. Symeonidis "npm-miner: An Infrastructure for Measuring the Quality of the npm Registry", MSR '18: 15th International Conference on Mining Software Repositories, Gothenburg, Sweden/10.1145/3196398.3196465

[23] Xin Yang, Hui Li, and Han Wang "NPM: An Anti-attacking Analysis Model of the MTD system Based on Martingale Theory", 978-1-5386-6950-1/18/2018 IEEE Symposium on Computers and Communications.

[24] Ruturaj K. Vaidya, Lorenzo De Carli, Drew Davidson, Vaibhav Rastogi "Security Issues in Language-based Sofware Ecosystems", arXiv:1903.02613v1 [cs.CR] 6 Mar 2019.

[25] Ahmed Zerouali, Tom Mens, Jesus Gonzalez-Barahona, Alexandre Decan, Eleni Constantinou, Gregorio Robles "A formal framework for measuring technical lag in component repositories — and its application to npm", J Softw Evol Proc. 2019;e2157/10.1002/smr.2157

[26] Terry Sun, Sam Rossi "js.rs – A Rustic JavaScript Interpreter", white paper.

[27] Pierre Chifflier, Geoffroy Couprie "Writing parsers like it is 2017", DOI 10.1109/SPW.2017.39, 2017 IEEE Symposium on Security and Privacy Workshops

[28] Steve Klabnik and Carol Nichols "The Rust Programming Language", 2015

[29] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman "Compilers: Principles, Techniques, and Tools, 2nd Edition", 2006

[30] Myoungkyu Song, Eli Tilevich "Systematic adaptation of dynamically generated source code via domain-specific examples", ISSN 1751-8806/ 10.1049/iet-sen.2016.0211

[31] Gebrehiwet B. Welearegai, Max Schlueter, Christian Hammer "Static Security Evaluation of an Industrial Web Application", 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)/10.1145/3297280.3297471

[32] Paulo Nunes, Iberia Medeiros, Jose C. Fonseca, Nuno Neves, Miguel Correia and Marco Vieira "Benchmarking Static Analysis Tools for Web Security", IEEE TRANSACTIONS ON RELIABILITY, VOL. 67, NO. 3, SEPTEMBER 2018/10.1109/TR.2018.2839339 /

[33] Micha Reiser, Luc Bläser "Accelerate JavaScript Applications by Cross-Compiling to WebAssembly", Proceedings of ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'17) 10.1145/3141871.3141873

[34] Vlad A Ionescu,Fabian Yamaguchi, Chetan Conikee, Manish Gupta "System and method for application security profiling", United States Patent Application Publication, Pub . No . : US 2018 / 0349614 A1, Dec . 6 , 2018