Mismorphism: The Heart of the Weird Machine

Prashant Anantharaman[1(✉)], Vijay Kothari[1], J. Peter Brady[1], Ira Ray Jenkins[1], Sameed Ali[1], Michael C. Millian[1], Ross Koppel[3], Jim Blythe[2], Sergey Bratus[1], and Sean W. Smith[1]

[1] Dartmouth College, Hanover, NH, USA
pa@cs.dartmouth.edu
[2] Information Sciences Institute, University of Southern California, Los Angeles, CA, USA
[3] Sociology Department, University of Pennsylvania, Philadelphia, PA, USA

Abstract. Mismorphisms—instances where predicates take on different truth values across different interpretations of reality (notably, different actors' perceptions of reality and the actual reality)—are the source of weird instructions. These weird instructions are tiny code snippets or gadgets that present the exploit programmer with unintended computational capabilities. Collectively, they constitute the weird machine upon which the exploit program runs. That is, a protocol or parser vulnerability is evidence of a weird machine, which, in turn, is evidence of an underlying mismorphism. This paper seeks to address vulnerabilities at the mismorphism layer.

The work presented here connects to our prior work in language-theoretic security (LangSec). LangSec provides a methodology for eliminating weird machines: By limiting the expressiveness of the input language, separating and constraining the parser code from the execution code, and ensuring only valid input makes its way to the execution code, entire classes of vulnerabilities can be avoided. Here, we go a layer deeper with our investigation of the mismorphisms responsible for weird machines.

In this paper, we re-introduce LangSec and mismorphisms, and we develop a logical representation of mismorphisms that complements our previous semiotic-triad-based representation. Additionally, we develop a preliminary set of classes for expressing LangSec mismorphisms, and we use this mismorphism-based scheme to classify a corpus of LangSec vulnerabilities.

1 Introduction

Mismatches between the perceptions of the designer, the implementor, and the user often result in protocol vulnerabilities. The designer has a high-level vision for how they believe the protocol should function, and this vision guides the creation of the specification. In practice, the specification may diverge from the

J. Anderson et al. (Eds.): Security Protocols 2019, LNCS 12287, pp. 113–124, 2020.
https://doi.org/10.1007/978-3-030-57043-9_11

initial vision due to real-world constraints, e.g., hardware or real-time requirements. The implementor then produces code to meet the specification based on their perceptions of how the protocol should function and, in some cases, how the user will interact with it. However, incorrect assumptions may produce vulnerabilities in the form of bugs or unintended operation. A user—informed by their own assumptions and perceptions—may then interact with a system or service that relies upon the protocol. A misunderstanding of the protocol and its operation can drive the user toward a decision that produces an unintended outcome. Ultimately, the security of the protocol rests on the consistency between the various actors' mental models of the protocol, the protocol specification, and the protocol implementation. A *mismorphism* refers to a mapping between different representations of reality (e.g., the distinct mental model of the protocol designer, the protocol implementor, and the end user) for which properties that ought to be preserved are not. In the past, we have used this concept and an accompanying semiotic-triad-based model to succinctly express the root causes of usable security failures [25]. We now apply this model to protocol design, development, and use. As mentioned earlier, many vulnerabilities stem from a mismatch between different actors' representations of protocols and the protocol operation in practice, e.g., the HeartBleed [11] vulnerability embodies a mismatch between the protocol specification, which involved validating a length field, and the implementation, which failed to do so. Therefore, it is natural to adopt the mismorphism model to examine the root causes of protocol vulnerabilities. That is precisely what we do in this paper.

In this paper, we examine protocol vulnerabilities and the mismorphisms upon which they are rooted. We develop a logic to express these mismorphisms, which enables us to capture the human mismatches that produce in vulnerabilities in code. Finally, we use this logical formalism to catalog the underlying mismorphisms that produce real-world vulnerabilities.

2 Related Work

The notion of mismorphism closely parallels the views expressed by Bratus et al. [8] in their discussion of exploit programming:

> "Successful exploitation is always evidence of someone's incorrect assumptions about the computational nature of the system—in hindsight, which is 20-20."

The mindset embodied in this quote forms the foundation for the field of language-theoretic security (LangSec).[1] Exploitation is unintended computation

[1] We only give a brief primer of LangSec in this paper. For those who are interested in learning more we recommend consulting the LangSec website [7].

performed on a *weird machine* [24] that the target program harbors. [2] Weird machines comprise the gadgets within a program that offer the adversary unintended computational capabilities to carry out an attack, e.g., NOP sleds, buffer overflows. Weird machines were never intended by protocol designers or implementors but organically arose within the protocol design and development phases. For instance, consider a designer who intends to design a web server program. The implementor of the software attempts to sanitize the input, but unwittingly lets malicious input through. This enables the adversary to supply unexpected input, resulting in unexpected behavior. The adversary repeatedly observes instances of unspecified program behaviour and uses these observations to craft an exploit program that they run on the exposed weird machine; this weird machine may serve as a Turing machine or otherwise expressive machine for the supplied input, the exploit program. To the designer, it was just a web server program; to the exploit programmer it provides an avenue of attack.

LangSec advocates taking a principled approach—one that is informed by language theory, automata theory, and computability theory—to parser design and development to eliminate the weird machine. LangSec facilitates the construction of safer protocols that behave closer to the way that designers and implementors envision by identifying best practices for protocol construction, such as parsing the input in full before program execution and ensuring the parser obeys known computability boundaries for safer computation. It also delivers tools such as Hammer and McHammer to achieve these goals [15,20].

Momot et al. [18] created a taxonomy of LangSec anti-patterns and used it to suggest ways to improve the Common Weakness Enumeration (CWE) database. Erik Poll [22] takes a different approach, classifying input vulnerabilities into two broad categories—flaws in processing input and flaws in forwarding input—and discusses examples from both categories in detail.

Pieczul and Foley studied Apache Struts code over a period of 12 years and systematically analyzed code changes and the nature of the security vulnerabilities reported [21]. They observed interesting phenomena pertaining to vulnerabilities and code security—and they developed a phenomena lifecycle that captures how these phenomena appear alongside developer awareness about the security problem.

We build upon this prior research, viewing mismorphisms as precursors to the weird machine. Our work is motivated by the belief that identifying and categorizing the mismorphisms that produce weird machines is a valuable step in systematically unpacking the causes of vulnerabilities and ultimately addressing them.

[2] For the reader interested in learning more about weird machines: Dullien [10] provides a formal definition for understanding weird machines and shows that it is feasible to build software that is resilient to memory corruption. Bratus and Shubina [9] also present exploit programming as a problem of code reuse, discuss how the adversary uses code presented by the weird machine to carry out the exploit, and describe colliding actors' abstractions of how the code works.

3 A Logic for Mismorphisms

Here, we provide a very brief primer on mismorphisms and present a logical model for capturing them. Our work here builds upon our earlier efforts to build a semiotic-triad-based mismorphism model [25], which was, in turn, inspired by early semiotics work by pioneers Ogden and Richards [19]. The logical representation presented here blends temporal logic with the idea of multiple interpreters. Following this section, we demonstrate how this logical model can be used to classify underlying causes of LangSec vulnerabilities by providing a preliminary classification using real-world examples.

In the context of this paper, a mismorphism refers to a difference in interpretations between two or more interpreters. That is, we can think of different interpreters (e.g., people) interpreting propositions or predicates about the world. In general, it is good when the interpretations agree and are in accordance with reality. However, when a predicate takes on a truth value under one interpretation but not another interpretation, we have a mismorphism, which may be a cause for concern. In our earlier applications, we found these mismorphisms were useful in understanding and classifying usable security failures and user circumvention. Here, we apply them to protocol and parser security. As mismorphisms deal with interpretations of predicates and how interpretations differ between interpreters, it's easy to see why formal logic provides a natural foundation to represent them.

We use the words *predicate* and *interpretation* in similar—albeit, not identical—manners to the common formal-logic meanings, e.g., as presented by Aho and Ullman [6]. However, instead of a binary logic, we use a ternary logic similar to Kleene's ternary logic [13,16].[3] We refer to a predicate as a function of zero or more variables whose codomain is $\{T, F, U\}$ where T is true, F is false, and U is uncertain/unknown. We refer to an *interpretation* of a predicate as an assignment of values (which may include U) to variables, which results in the predicate being interpreted as T, F, or U. A predicate is interpreted as T if after substituting all variables for their truth values, the predicate is determined to be T; it is interpreted as F if after substituting all variables for their truth values, the predicate is determined to be F; if we are unable to determine whether the predicate is T or F, the predicate is interpreted as U.

The interpretation must be done by someone (or perhaps something) and that someone is the *interpreter*. In this paper, common *interpreters* include the oracle O who interprets the predicate as it is in reality, the designer D, the implementor I, and the user Y. We note that some interpreters may not have adequate information to assign precise values to the variables that ensure the predicate is interpreted as T or F. It is in these instances that the predicate may be interpreted as U. In this paper, we use $P|_A$ to denote the interpretation of predicate P by interpreter A.

To represent mismorphisms we need a way to express the relationship between interpretations of a predicate. Thus, we have the following:

[3] We do not specify a specific ternary logic system for evaluating predicates in this paper.

[*Predicate*] [*Interpretation Relation*] [*List of interpreters*]

The interpretation relations over the set of all predicate-interpreter pairs are k-ary relations where $k >= 2$ is the number of interpreters there are in the interpretation relation—and each k-ary relation is over the interpretations of the predicate by the k interpreters. The three interpretation relations we are concerned with in this paper are: the interpretation-equivalence relation ($\underset{\text{interp.}}{=}$), the interpretation-uncertainty relation ($\underset{\text{interp.}}{\stackrel{?}{=}}$), and the interpretation-inequivalence relation ($\underset{\text{interp.}}{\neq}$).[4] These relations are defined as follows, where each P represents a predicate and each A_i represents an interpreter:

- $P \underset{\text{interp.}}{=} A_1, A_2, \ldots A_k$ if and only if P, as interpreted by each A_i, has a truth value that's either T or F (never U)—and all interpretations yield the same truth value.
- $P \underset{\text{interp.}}{\stackrel{?}{=}} A_1, A_2, \ldots A_k$ if and only if P takes on the value U when interpreted by at least one A_i.
- $P \underset{\text{interp.}}{\neq} A_1, A_2, \ldots A_k$ if and only if P interpreted by A_i is T and P interpreted by A_j is F for some $i \neq j$.

There are a few important observations to note here. One is that the oracle O always holds the correct truth value for the predicate by definition. Another is that if we only know the $\underset{\text{interp.}}{\stackrel{?}{=}}$ relation applies, we won't know which interpreter is uncertain about the predicate or even how many interpreters are uncertain unless $k = 2$ and one interpreter is the oracle. Similarly, if we only know that the $\underset{\text{interp.}}{\neq}$ relation applies, we do not know where the mismatch exists unless $k = 2$. That said, knowledge that the oracle O always holds the correct interpretation combined with other facts can help specify where the uncertainty or inequivalence stems from. Last, the $\underset{\text{interp.}}{=}$ relation will not be applicable if either the $\underset{\text{interp.}}{\stackrel{?}{=}}$ or the $\underset{\text{interp.}}{\neq}$ interpretations are applicable; however, $P \underset{\text{interp.}}{\stackrel{?}{=}} A_1, \ldots A_k$ and $P \underset{\text{interp.}}{\neq} A_1, \ldots A_k$ can both be applicable simultaneously.

The purpose of creating this model was to allow us to capture mismorphisms. Mismorphisms correspond to instances where either the interpretation-uncertainty relation or interpretation-inequivalence relation apply.

It may be valuable to consider some natural extensions to this logical formalism. In select cases, we may consider multiple interpreters of the same role. In these instances, we assign subscripts to distinguish roles, e.g., D, I_1, I_2, O. Also,

[4] Note that for $k = 2$, if we confine ourselves to predicates that take on only T or F values, the relation $\underset{\text{interp.}}{=}$ is an equivalence relation in the mathematical sense, as one might expect, i.e., it obeys reflexivity, commutativity, and transitivity.

there are temporal aspects that may be relevant. Predicates can be functions of time and so can the interpretations. While we use the common $v(t)$-style notation to represent a variable as a function of time within a predicate, in select cases we consider the interpreter as a function of time, e.g., $I_4^{t_3}$ means the interpretation is done by implementor I_4 at time $t = t_3$.

4 Preliminary Classification

Here, we describe various classes of mismorphisms using the mathematical notation we just developed. All the vulnerabilities we catalog fall into one of the categories of mismorphisms we describe below.

4.1 Failed Assumption of Language Decidability

Any input language format needs to be decidable for the implementor to be able to parse and make sure that there are no corner cases when the program can enter unexpected states or fail to terminate. When the designer assumes a language L is decidable (in the absence of proof, that it is), the program may harbor the potential for unexpected computation.

For one example, the Ethereum platform uses a Turing-complete input language to enable its smart contracts. It is demonstrably more difficult to build a parser for such an input language. Such added complexity led to the Ethereum DAO disaster [27], in which all ethers were stolen, forcing the developers to perform a highly controversial hard fork. As a result, some developers built a decidable version of Solidity called vyper [12].

We define such a language mismorphism by the form:

$$L \text{ is decidable} \underset{\text{interp.}}{\overset{?}{=}} O, D \tag{1}$$

A diagrammatic representation of the above formalism can be found in Fig. 1.

4.2 Shotgun Parsers

Shotgun parsers perform input data checking and handling interspersed with processing logic. Shotgun parsers do not perform full recognition before the data is processed. Hence, implementors may assume that a field x has the same value at time t and time $t + \delta$, but the processing logic may change the value of the field x in an input buffer B.

This mismorphism relation is seen below:

$$B(t) = B(t + \delta) \underset{\text{interp.}}{\neq} O, I \tag{2}$$

Implementors may expect the buffer to be intact across time, but that is not observed to be the case. Shotgun parsing can cause mismorphisms in two distinct ways. First, a partially validated input may be wrongly treated as though

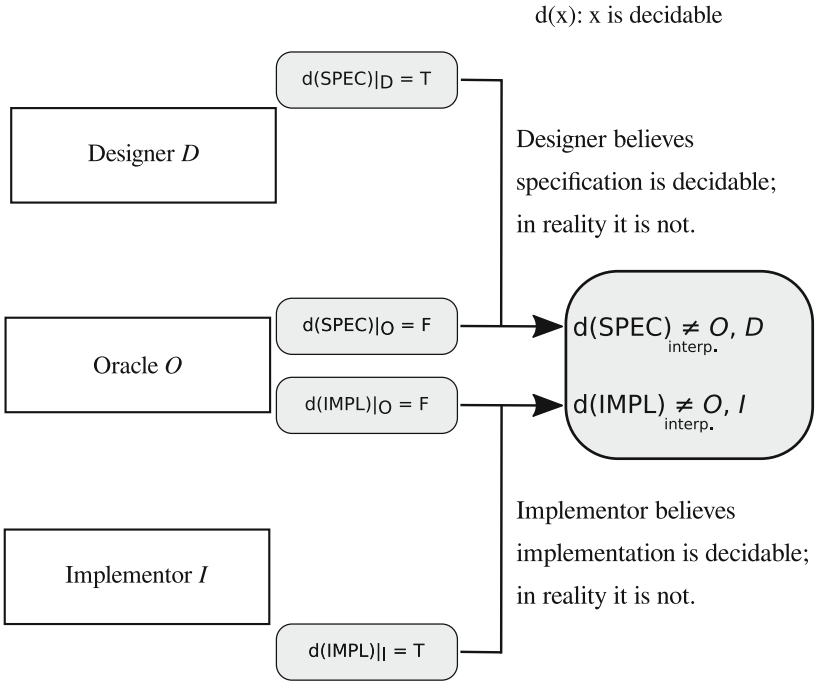

Fig. 1. One class of mismorphism where implementors and designers both disagree with the reality that the language is actually undecidable.

it is fully validated. Suppose Implementor 1 performs the shotgun parsing and knoww the input to be only partially validated. Then, Implementor 2 works on execution and assumes the input is fully validated by the time the code segment is executed. This type of a shotgun parser mismorphism can be represented as follows:

$$B \text{ is accepted} \underset{\text{interp.}}{\neq} I_1, I_2 \tag{3}$$

Second, the same implementor may perform shotgun parsing and be responsible for working on the execution code. But they may interpret the same protocol differently during those different times. This type of a mismorphism can be represented as follows:

$$B \text{ is accepted} \underset{\text{interp.}}{\neq} I^{t_1}, I^{t_2} \tag{4}$$

4.3 Parser Inequivalence for the Same Protocol

Designers of protocols intend for two endpoints to have the exact same functionality, and build identical parse trees. The Android Master Key bug [14] is an apt

example for this type of a mismorphism. The parsers for the unzipping function in Java and C++ were not equivalent, leading to a parsing differential.

We describe this relation as:

$$\text{Parsers } P_1, P_2 \text{ are equivalent} \underset{\text{interp.}}{\neq} O, D, I \tag{5}$$

4.4 Implementor Is Unaware that Some Fields Must Be Validated

Designers of protocols introduce new features in the specification of the protocol without describing them fully or accurately. The designer introduces a field x in the protocol, but the interpreter does not entirely understand how to interpret it. The Heartbleed vulnerability [11] was an example of this. The designers included the heartbeat message, but the implementors did not completely understand it and missed an additional check to make sure the length fields matched.

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I \tag{6}$$

4.5 Types of Fields in Buffer Are Not Fixed During Buffer Life Cyle

The types of values that have already been parsed must remain constant. Sometimes, implementors assume field x is treated as type $t(x)$ throughout execution. In reality, the field may be treated as a different type at certain points during execution.

$$type(x) \text{ is fixed} \underset{\text{interp.}}{\neq} O, D, I \tag{7}$$

5 A Catalog of Vulnerabilities and Their Mismorphisms

Below, we provide a small catalog of some vulnerabilities and the mismorphisms that we believe produced them.

Shellshock: Bash unintentionally executes commands that are concatenated to function definitions that are inside environment variables [17].

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$$

The sanity check C here makes sure that once functions are terminated, the variable shouldn't be reading commands that follow it.

Rosetta Flash: SWF files that are requested using JSONP are incorrectly parsed once they are compressed using *zlib*. Compressed SWF files can contain only alphanumeric characters [26].

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$$

The specification of the SWF file format is not exhaustively validated using a grammar. The fix uses conditions such as checking for the first and last bytes for special, non-alphanumeric characters.

Heartbleed: The protocol involves two length fields, one that specifies the total length of the heartbeat message; the other specifies the size of the payload of the heartbeat message [11].

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$$

Sanity check C involves verifying the length fields l_1 and l_2 match.

Android Master Key: The Java and C++ implementations of the cryptographic library performing unzipping were not equivalent [14].

$$\text{Parsers } P_1, P_2 \text{ are equivalent} \underset{\text{interp.}}{\neq} O, D, I$$

Ruby on Rails - Omakase. The Rails YAML loader doesn't validate the input string and check that it is valid JSON. And it doesn't load the entire JSON; instead, it just starts replacing characters to convert JSON to YAML [23].

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$$

Sanity check C should first recognize and make sure the JSON is well-formed, before replacing the characters in YAML.

Nginx HTTP Chunked Encoding: Large chunk size for the Transfer-Encoding chunk size trigger integer signedness error and a stack-based buffer overflow [2].

$$B \text{ is accepted} \underset{\text{interp.}}{\neq} I_1, I_2$$

The shotgun parser works on execution without validating the value of the length field, which could be much larger than allowed, thereby causing buffer overflows. All implementors must work with the same knowledge, and the input must first be recognized fully.

Elasticsearch Crafted Script Bug: Elasticsearch runs Groovy scripts directly in a sandbox. Attackers were able to craft a script that would bypass the sandbox check and execute shell commands [4].

$$L \text{ is decidable} \underset{\text{interp.}}{\overset{?}{=}} O, D$$

Developers of Elasticsearch had to explore the option of abandoning Groovy in favor of a safe and less dynamic alternative.

Mozilla NSS Null Character Bug: When domain names included a null character, there was a discrepancy between the way certificate authorities issued certificates and the way SSL clients handled them. Certificate authorities issued certificates for the domain after the null character, whereas the SSL clients used the domain name ahead of the null character [1].

$$\text{Parsers } P_1, P_2 \text{ are equivalent} \underset{\text{interp.}}{\neq} O, D, I$$

Although having a null character in a certificate is not accepted behavior, certificate authorities and clients do not want to ignore requests that contain them. So they follow their own interpretations, resulting in a parser differential.

Adobe Reader CVE-2013-2729: In running length encoded bitmaps, Adobe Reader wrote pixel values to arbitrary memory locations since there was a bounds check that was skipped [3].

$$B \text{ is accepted} \underset{\text{interp.}}{\neq} I_1, I_2$$

The code used a shotgun parser where the implementor of the processing logic assumed all fields were validated. The bounds check was never performed.

OpenBSD - Fragmented ICMPv6 Packet Remote Execution: Fragmented ICMP6 packets cause an overflow in the mbuf data structure in the kernel may cause a kernel panic or remote code execution depending on packet contents [5].

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$$

Implementors of the ICMP6 packet structures in OpenBSD did not understand how to map it to the existing `mbuf` structure, and then validate it.

6 Conclusion

In this paper, we proposed a novel approach to categorizing the root causes of protocol vulnerabilities. We created a new logical model to express mismorphisms, grounded in the semiotic-triad based representation of mismorphisms explored in our earlier work. We then used this logical model to develop a preliminary set of mismorphism classes for capturing LangSec vulnerabilities. Finally, we created a small catalog of vulnerabilities and demonstrated how our classification scheme could be used to classify the mismorphisms those vulnerabilities embody.

Acknowledgement. This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-16-C-0179 and Department of Energy under Award Number DE-OE0000780.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force, DARPA, United States Government or any agency thereof.

References

1. CVE-2009-3555 The Mozilla Network Security Services (NSS) fails to properly validate the domain name in a signed CA certificate, allowing attackers to substitute malicious SSL certificates for trusted ones. Available from Vulners. https://vulners.com/exploitdb/EDB-ID:26703
2. CVE-2013-2028 Nginx HTTP Server 1.3.9-1.4.0 Chunked Encoding Stack Buffer Overflow. Available from Rapid 7. https://www.rapid7.com/db/modules/exploit/linux/http/nginx_chunked_size
3. CVE-2013-2729 Adobe Reader X 10.1.4.38 - BMP/RLE heap corruption. Available from Vulners. https://vulners.com/exploitdb/EDB-ID:26703
4. CVE-2015-1427 The Groovy scripting engine in Elasticsearch before 1.3.8 and 1.4.x before 1.4.3 allows remote attackers to bypass the sandbox protection mechanism and execute arbitrary shell commands via a crafted script. Available from Vulners. https://vulners.com/cve/CVE-2015-1427
5. OpenBSD's IPv6 mbufs remote kernel buffer overflow. Available from Vulners. https://vulners.com/cert/VU:986425
6. Aho, A., Ullman, J.: Foundations of Computer Science: C Edition, Chapter 14, July 1994. http://infolab.stanford.edu/~ullman/focs.html
7. Bratus, S.: LANGSEC: Language-theoretic Security: "The View from the Tower of Babel". http://langsec.org
8. Bratus, S., Locasto, M., Patterson, M., Sassaman, L., Shubina, A.: Exploit programming: from buffer overflows to "Weird Machines" and theory of computation. Login USENIX Mag. **36**(6), 13–21 (2011)
9. Bratus, S., Shubina, A.: Exploitation as code reuse: on the need of formalization. IT-Inf. Technol. **59**(2), 93–100 (2017). https://doi.org/10.1515/itit-2016-0038
10. Dullien, T.F.: Weird machines, exploitability, and provable unexploitability. IEEE Trans. Emerg. Top. Comput. (2017). https://doi.org/10.1109/TETC.2017.2785299
11. Durumeric, Z., et al.: The Matter of Heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference, pp. 475–488. ACM (2014). https://doi.org/10.1145/2663716.2663755

12. Ethereum: Pythonic Smart Contract Language for the EVM. https://github.com/ethereum/vyper
13. Fitting, M.: Kleene's three valued logics and their children. Fundam. Inf. **20**(1–3), 113–131 (1994). http://dl.acm.org/citation.cfm?id=183529.183533
14. Freeman, J.: Exploit (& Fix) Android "Master Key". http://www.saurik.com/id/17
15. Hermerschmidt, L.: McHammerCoder: a binary capable parser and unparser generator, https://github.com/McHammerCoder/McHammerCoder
16. Kleene, S.C.: Introduction to metamathematics (1954)
17. Mary, C.: Shellshock attack on linux systems-bash. Int. Res. J. Eng. Technol. **2**(8), 1322–1325 (2015)
18. Momot, F., Bratus, S., Hallberg, S.M., Patterson, M.L.: The seven turrets of babel: a taxonomy of LangSec errors and how to expunge them. In: 2016 IEEE Cybersecurity Development (SecDev), pp. 45–52, November 2016. https://doi.org/10.1109/SecDev.2016.019
19. Ogden, C.K., Richards, I.A.: The Meaning of Meaning: A Study of the Influence of Language upon Thought and of the Science of Symbolism. Harcourt Brace and Company, San Diego (1927)
20. Patterson, M.: Parser combinators for binary formats, in C. https://github.com/UpstandingHackers/hammer
21. Pieczul, O., Foley, S.N.: The evolution of a security control. In: Anderson, J., Matyáš, V., Christianson, B., Stajano, F. (eds.) Security Protocols 2016. LNCS, vol. 10368, pp. 67–84. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62033-6_9
22. Poll, E.: LangSec revisited: input security flaws of the second kind. In: 2018 IEEE Security and Privacy Workshops (SPW), pp. 329–334. IEEE (2018). https://doi.org/10.1109/SPW.2018.00051
23. Rezvina, S.: Rails' Remote Code Execution Vulnerability Explained. https://codeclimate.com/blog/rails-remote-code-execution-vulnerability-explained
24. Shapiro, R., Bratus, S., Smith, S.W.: "Weird Machines" in ELF: a spotlight on the underappreciated metadata. In: Proceedings of the 7th USENIX Conference on Offensive Technologies. WOOT 2013, USENIX Association, Berkeley, CA, USA (2013). http://dl.acm.org/citation.cfm?id=2534748.2534763
25. Smith, S.W., Koppel, R., Blythe, J., Kothari, V.: Mismorphism: a semiotic model of computer security circumvention. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, p. 25. ACM (2015)
26. Spagnuolo, M.: Abusing JSONP with rosetta flash. https://miki.it/blog/2014/7/8/abusing-jsonp-with-rosetta-flash/
27. Torpey, K.: The DAO disaster illustrates differing philosophies in bitcoin and ethereum. https://www.coingecko.com/buzz/dao-disaster-differing-philosophies-bitcoin-ethereum