What We Have Here Is Failure to Validate: Summer of LangSec

Sameed Ali, Prashant Anantharaman, Zephyr Lucas, and Sean W. Smith | Dartmouth College

This article reviews several exploits of input-handling vulnerabilities and categorizes these vulnerabilities based on their root causes. We describe the language-theoretic security paradigm and discuss how it can be used to tackle these categories of vulnerabilities.

In the summer of 2020, we saw a surge in exploits of input-handling vulnerabilities across various systems from the modern to the more than 20-year-old Treck TCP/IP stack. Although such vulnerabilities have been a commonly exploited attack vector since the 1980s, they are still prevalent. Languages (such as C, C++, and assembly) that lack built-in memory protections are incredibly prone to bugs stemming from insufficient or no input validation. One approach to prevent this class of vulnerabilities is to apply a more formally rigorous process to input handling, such as Language-Theoretic Security (LangSec). In this article, we study 10 vulnerabilities and exploits from the summer of 2020 through the lens of LangSec. We believe that LangSec can be a useful tool to prevent such exploits in the future.

The LangSec approach advocates defining various input formats as formal grammars and building recognizers for these grammars to *decide* whether an input is valid, as proposed in 2013 by Sasseman et al.[11] This framework applies to various network protocols and file formats that require complex parsers to process input. LangSec provides a rigorous methodology to implement these parsers in a way that would minimize parsing errors.

The vulnerabilities we study in this article fall under three broad categories, as follows:

- when the parser does not reject invalid input
- when the specification or grammar describes a proper superset of the input that the program can handle correctly
- when two parsers are expected to consume the same grammar but behave differently.

Flawed and Insufficient Parsers: Basic LangSec Failures

Figure 1 shows the basic LangSec antipattern, in which flawed code implicitly assumes the input is correctly formatted and acts on invalid input as a result. The three vulnerabilities described next fit this antipattern.

Ripple20

June 2020 brought news of Ripple20, a set of vulnerabilities in the Treck TCP/IP stack. This TCP/IP stack is over 20 years old but used in "hundreds of millions" of embedded systems, including critical infrastructure such as power, aircraft, and health care.[2] Besides the direct threat to the large number of compromised systems themselves, Ripple20 has additional high-impact downstream consequences because it is a networking stack that, by definition, exposes an interface to the outside world. Consequently, Ripple20 vulnerabilities may enable adversaries to read and write data on the devices, as well as execute code, and thus compromise other systems. Remediation will be difficult as real-world embedded systems are notoriously hard to track down and patch.

Digital Object Identifier 10.1109/MSEC.2021.3059167
Date of current version: 5 March 2021

Two of Ripple's vulnerabilities (i.e., "#1" and "#3" in Kol et al. [7]) are straightforward failures by the Domain Name System (DNS) resolver to validate external input.

- In the first vulnerability, flawed code constructs a requested hostname and copies it into a buffer. However, the adversary-supplied MX record provides both the `RDLENGTH` value used to calculate the size of the buffer and the data from which the requested hostname is constructed. As a result, the adversary can supply a record that tricks the code into corrupting its heap by copying a large byte sequence into a too-small buffer. Correctly behaving code should have rejected this malformed record.
- In the second vulnerability, the flawed code does not correctly confine reads to data relevant to the adversary-supplied record—so a malformed record can trick the code into copying internal data to an output buffer.

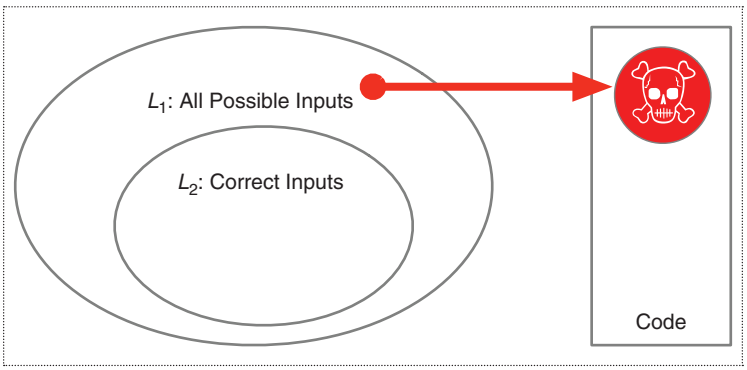

Figure 1. In the basic antipattern, the flawed code implicitly assumes the input is in L_2 but does not check; attacks are possible when the adversary crafts an input in $L_1 \backslash L_2$.

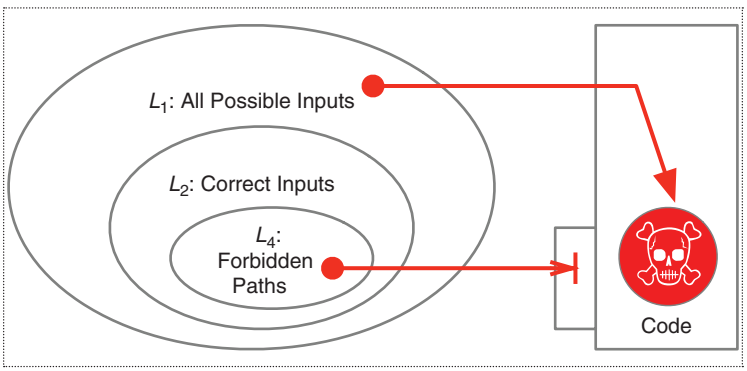

Figure 2. In a variation of the basic antipattern, the flawed code checks for disallowed requests (L_4) but implicitly assumes these requests are in L_2; the adversary can craft inputs in $L_1 \backslash L_2$ that bypass the checks but get interpreted as something in L_4.

The Thales Module

August brought news of a vulnerability in a Thales module potentially used in "billions of things."[8] At its core, this vulnerability manifests as a variation on the basic LangSec antipattern. The variation is shown in Figure 2. In the case of the Thales module, flawed code determines access control by checking if an input parameter specifies something forbidden—but this test implicitly assumes the input parameter is correctly formatted (i.e., a path with a single slash). If the adversary goes outside the safe input space by supplying what would be a forbidden parameter, except using two slashes instead of one, the flawed code accepts the parameter and then ignores the second slash.

How LangSec Can Fix These Problems

In these examples based on exploiting the basic LangSec antipattern, the software programs expected their input to follow certain rules and the attacker capitalized on these expectations by providing data that was not structured according to these rules but processed nonetheless. This in turn caused unexpected and harmful computation to be performed. LangSec calls for these input rules to be formally described as a grammar. Describing the input rules as a grammar allows a verifier to parse any potential input to see if it adheres to these rules before passing it on for processing. In the Ripple20 vulnerabilities, a parser would have recognized malformed records and rejected them before any harmful computation would have been done. Similarly, in the Thales Module, an input parser would have rejected input parameters with excess slashes, thereby protecting the code from the forbidden paths.

The Official Grammar Is Wrong: When Specifications are Incorrect

The standard LangSec defense pattern to the problems of Figures 1 and 2 is to formally specify an expected input language, L_2, and to test whether an input is in L_2 before acting on it. However, this approach raises the question (depicted in Figure 3): just because an input is in the language the developer expects, does that it mean the code will handle it safely? This section considers some recent vulnerabilities for which the answer to the question (above) is "no."

Summer 2020 saw many flaws arising because the set of correctly handled inputs are only a proper subset of the inputs within the validation grammar. This can stem from either an error in formally describing the intended input or from improperly handling all intended inputs.

The Wallpaper of Death

June brought news of the Android "Wallpaper of Death."[1] A particular picture, when set as the background image

for a Samsung or Google Pixel (or other phones that use the default Android color engine), will softlock the phone by causing it to endlessly reboot. The picture contains a pixel whose color values cause an overflow error when the phone is calculating the luminescence value that the OS handles by rebooting the phone. During the reboot process, the background image is loaded, overflows, and initiates another reboot. This renders the phone unusable until the user boots the phone in safe mode and deletes the image, resetting the phone to the default background.

This raises some interesting LangSec questions as the image in question uses a properly formatted red, green, blue (RGB) color space—but the phone first transforms the RGB to sRGB, and the transform code can turn some valid RGB images (such as this one) into illegally formatted sRGB images.

Non-Click iOS Email 0 day

In April 2020, ZecOps reported that they found a zero-click attack on iPhone and iPad default email applications.[14] Users do not need to take any action; when the email shows up in their mailbox, it gives access to the attacker. The attacker can even delete the crafted email that gave them access. Such zero-click attacks are hence extremely difficult to detect. ZecOps reported that several top-level executives and journalists across the world might have been affected by this vulnerability.

ZecOps reported that correctly formatted (but deviously crafted) emails, such as the following examples, could trigger vulnerabilities:

- *Out of bounds write*: The email application calls the `ftruncate` syscall to truncate or extend a file to a specific length, but it does not check the error condition. With the right input, the application will write beyond the end of mapped space.
- *Heap-based overflow*: The email application tries to copy `0x200000` bytes into memory allocated using `mmap`. However, if this syscall fails, the application only allocates 8 bytes and copies over the much larger `0x200000` bytes, overflowing the heap.

The Snapdragon

August brought news of a large number of vulnerabilities within the Snapdragon digital signal processor used in many Android smartphones, including but not limited to Google, Samsung, OnePlus, Xiaomi, LG, Android and which are accessible by third-party applications.[10] Such devices have many separate processors; communication between them is handled by software generated by the Hexagon software development kit. However, as Slava Makkaveev discussed at DefCon,[15] this code fails to handle certain kinds of correctly (but oddly) formatted input: data and buffer lengths are treated as unsigned integers but tested as signed integers—so crafted input with a large enough length to be regarded as negative when interpreted as signed can result in heap overflow.

SigRed

Bastille Day (14 July 2020) brought news of "SigRed," a 17-year vulnerability in the Windows DNS. The bug enables remote adversaries to execute code on the victim machine, on "the DNS servers of practically every small and medium-sized organization around the world."[3] Researchers at Check Point[13] found that when the code handles certain kinds of records sent in response to a forwarded DNS query, it calculates the length of the record using a 16-bit unsigned integer. However, the adversary can use compression tricks to construct a valid record whose length exceeds 2.[16] Once more, the flawed code copies too many bytes into too small a buffer, resulting in heap overflow.

Ripple Again

Another one of the Ripple vulnerabilities demonstrates this pattern ("#2" in Kol et al.[7]). The flawed code uses a 16-bit unsigned integer to track a "label length" value, but correct inputs can be constructed whose label length overflows that. As a result, the code will allocate a too-small buffer, and then copy a much larger byte sequence into it. This can also be viewed as a parsing problem: The flawed code uses two different ways to calculate a length, and there are correct inputs for which these two approaches do not agree.

How LangSec Can Fix These Problems

In these examples, the problem was not a failure to validate, but a failure to validate with the *correct* specifications.

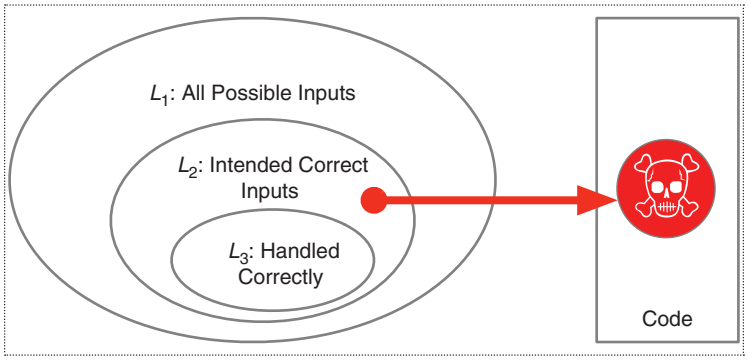

Figure 3. In many cases, the flawed code is intended to handle some target L_2 of valid inputs—but instead properly handles only a proper subset L_3. Attacks are possible when the adversary crafts an input in $L_2 \backslash L_3$. Simply validating against L_2 is not enough—we need tools to help ensure that $L_2 \backslash L_3$ is empty.

To fix problems like this, the grammar must be rewritten to match what is handled correctly, or the computation must be reworked to handle all of the expected input. We must check whether the input is within the specified grammar. However, we also need to be careful that the grammar we are checking against matches a program's "safe input." Therefore, we need to develop tools that can discover grammars for the application or, at minimum, be able to find potential errors with grammars that are considered "correct." Alternatively, if the legal inputs in $L_2 \backslash L_3$ have no legitimate use, the developers may wish to restrict the official input space and tighten the input filter accordingly.

LangSec on the Inside

Other Ripple Vulnerabilities

Other Ripple vulnerabilities[6] pertain not to code parsing input but, rather, to code parsing its own internal data structure (Figure 4). When processing tunneled and fragmented Internet Protocol packets, the code constructs an internal data structure with two different ways of representing the same length: an overall length in the initial part, and the sum of the lengths of each part. In "correct" configurations of this data structure, these match; however, correct inputs exist that can cause the flawed code to construct an internal state in which these values do not match, leading to adversarial-directed memory corruption.

How LangSec Can Fix These Problems

Just as it is important for software to ensure that its external inputs are correctly formatted before acting on them, it is also important to check its internal inputs and internal outputs. Indeed, the basic programming tenet of defining and preserving the *invariant* for a data structure can be seen in a LangSec context: Indicate precisely what one means as correct.

Differentials: Can Parsers Disagree?

Parsing Differential

In LangSec, the role of formal languages is not just confined to deciding whether an input is safe to act on. It is also key in *parsing*; i.e., how a program understands what an input means.

In the software world, the multiple programs often consume the same input—and there is an underlying assumption that different programmers read a specification and understand it the same way. Disagreements lead to *parser differentials* (Figure 5), perhaps documented initially in 2010 when Len Sassamen et al.[5] found that Public Key Certificate Authorities could grant servers to domains that were invalid and some application programming interfaces accepted these invalid certificates.

The Psychic Paper. May brought news of a parser differential vulnerability in iOS.[12] In the Apple Ecosystem, *property lists* (or *plists*) are used to store serialized data in various contexts, such as code signatures and configuration files. The list of properties the plist defines in code signatures comprise the *entitlements*. There are over 1,800 entitlements available for the latest versions of the iOS (the mobile operating system), whereas the Mac OSX versions support just fewer than 1,000 entitlements each. These entitlements also specify how the application interacts with the kernel. It can hold a list of drivers and system calls that the application can access.

The Psychic Paper vulnerability uses the fact that iOS uses "at least four" different XML parsers, and they have differentials. One example is how parsers handle comments in plists. Comments are usually enclosed in <!-- and --> tags, and everything enclosed within these tags is considered to be a comment. However, three of the iOS XML parsers (IOKit, CF, and XPC) handle the not-quite-correct comment delimiters 1) <!---> and 2) <!--> differently.

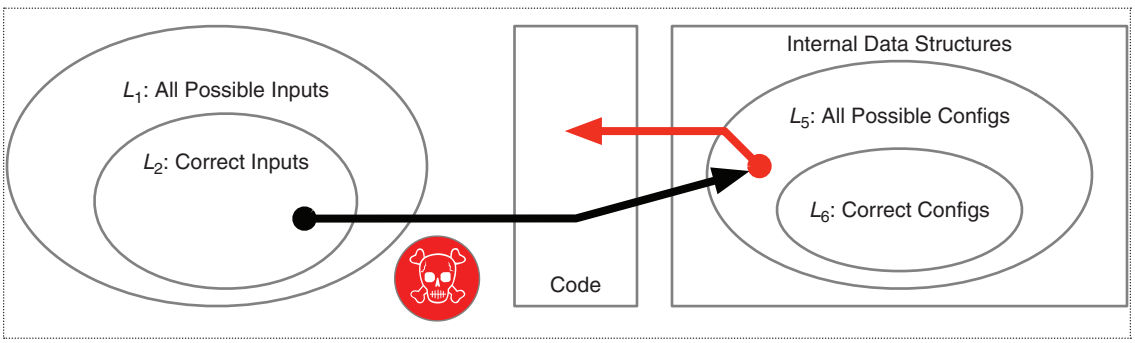

Figure 4. In some vulnerabilities, crafted but correct input can trick the flawed code into constructing an incorrectly formatted internal state; when the code later acts on that state, it goes wrong.

- The IOKit sees the <!– in 1) and thinks it is the start of a comment. It ends the comment when it sees the end of 2) <!––>.
- The CF parser has a bug that only scans two places after seeing the ! in 1). This means that a – is parsed twice and the first tag 1) opens a comment. The second tag 2) is treated as the opening of a new comment.
- XPC ignores both 1) and 2) but differs from the IOKit and CF in other constructs such as the usage of double-quotes, " and square brackets, [].

By being able to construct plists containing material that some parsers ignore as comments and others regard as real, the adversary can thus forge his or her own entitlements and can get access to any memory location the user can access and execute system calls and drivers. The adversary can also alter thread register states and read and write to process memory.

As mentioned previously, the problem stems from the fact that the three parsers handle comments differently. One way to deal with this could be to specify the context-free grammar needed to recognize plists, and then build a parser to recognize this grammar. Instead of a specification, if developers start with the formal grammar needed to recognize plists, parser differentials can be minimized.

How LangSec Can Fix These Problems. With parser differentials, the correct specifications are not the same across various implementations. Using parser generators that use a data-description language or grammar could ensure that the parsers implement the same grammar across different implementations. Another approach would be to use parser combinators. They implement code that looks like the grammar and can be easily verified. When different implementations use parser combinators, they can be visually verified to be equivalent, as well as equivalent to the grammar specification.

Semantic Differential

The Shadow. July also brought news of the Shadow attack[9], which might also be seen as a form of differential, albeit a semantic one. A PDF viewer can

1. render a PDF file as a "virtual piece of paper" to the user
2. allow a user to digitally sign the document seen by the user
3. allow a user to append an incremental update to the document
4. tell the user that a digital signature is valid.

The problem is that Behavior 3 could permit certain harmful types of incrementally saved changes after Behavior 2 and, hence, allow the attacker to configure the PDF so that the visible content of the document that the signer saw in Behavior 2 differs from the content the verifier sees in Behavior 4 (Figure 6). Attackers can construct a form, contract, or bill, for example, and get it approved via a digital signature. However, it is possible for the attacker to then modify the document to display content different from the one approved.

The problem is that there is a gap between the core meaning of the PDF file—the virtual piece of paper the user sees—and the parsed PDF structure. As Figure 7 shows, the verifier cannot (easily) distinguish between two different abstract syntax trees with significantly different signature semantics the same way. (This was also demonstrated with earlier versions of PDF long ago.[4])

How LangSec Can Fix These Problems. A common solution that many PDF viewers currently use is to remove any unused objects prior to signing; however, this stops only some flavors of the Shadow attack and not all of them. A stronger one,[9] would be to reject any document that contains a nonsigning incremental save after a signature. While this is effective, it is possibly too harsh, as

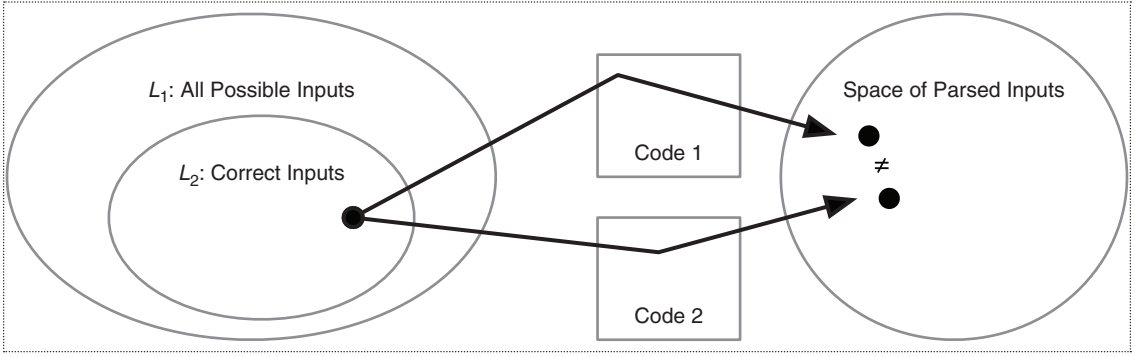

Figure 5. In *parser differentials*, two different programs can construct different parse trees—and take different actions—for the same input.

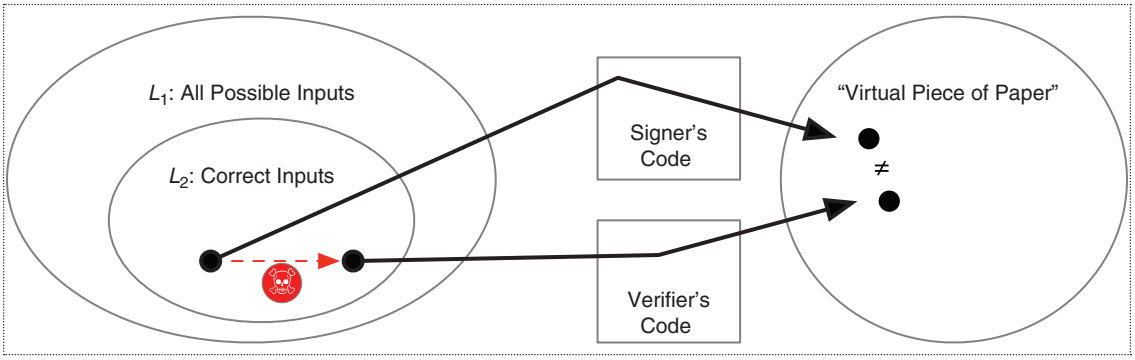

Figure 6. In the Shadow attack, the signer sees a "virtual piece of paper" and signs it; the verifier later sees a valid signature but a different virtual piece of paper. In between, the adversary modifies the PDF.

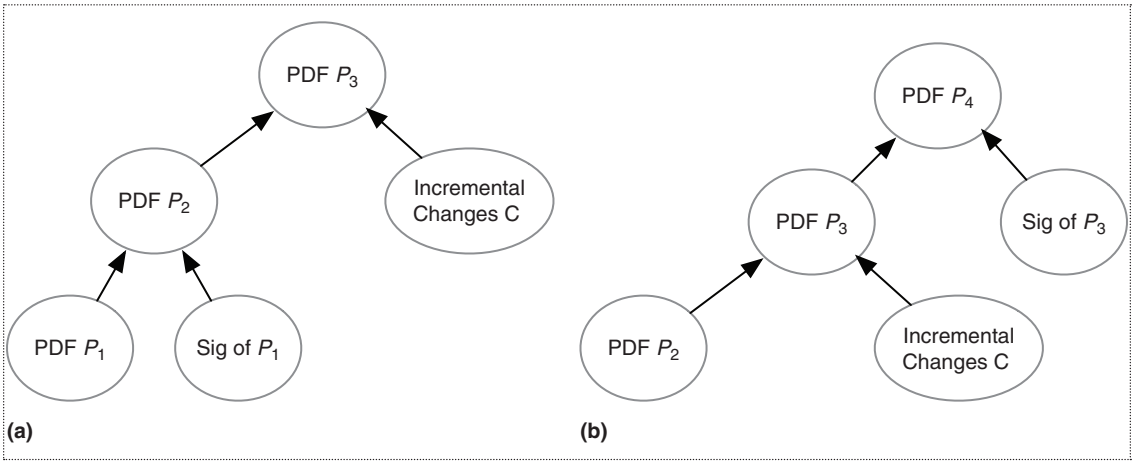

Figure 7. In this simplified diagram, the verifier may perceive that the signer signed P_3: the PDF P_1 modified by changes *C*. However, that only happened in (b); in (a), the signer only signed P_1.

there are use cases where changes made post signature are actually desired (e.g., filling out an approved form).

Work must be done to isolate exactly what is considered a "safe change"—and more deeply bind the action of producing/verifying a signature to the semantics of how the document is being presented to the user. Ideally, if two parse trees are semantically different, then the code consuming them should enable the user to distinguish the difference.

Although LangSec has been around since 2013, developers have not adopted it much as far as we know. No studies have been conducted to evaluate the usability of the several parser-combinator libraries, data-description languages, and parser generator tools available to enforce LangSec principles. We believe such a study would clarify why such tools are yet to receive traction.

Instrumenting existing software with LangSec parsers would increase code complexity and add a runtime overhead. However, we believe such overheads would be minimal with respect to large code bases, and LangSec parsers make code easy to audit since parsers are written like grammars. Several other categories of code vulnerabilities—such as use-after-free, null-pointer-dereference, and integer overflows—cannot be prevented using LangSec (unless, of course, they are tickled by illegal inputs). We must use LangSec in conjunction with several other security and memory protection tools to minimize exploits.

Summer 2020 reminded us that it remains important for software to specify and validate its input (e.g., via parser-combinator libraries) and for all consumers in an ecosystem to parse input the same way. However, we also see reminders that we need better tools to help discover when the set of inputs a program can safely handle is a strict subset of the set it was designed for. ■

Acknowledgments

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) under contracts HR001119C0075 and

HR001119C0121. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

1. B. Bhushan. "Why a single pixel on this wallpaper crashes Android phones?" SlashGear. June 2020. https://www.slashgear.com/why-a-single-pixel-on-this-wallpaper-crashes-android-phones-13624653/The%20explanation&text=The%20picture%20basically%20didn't,space%20that%20the%20image%20uses
2. C. Cimpanu. "Ripple20 vulnerabilities will haunt the IoT landscape for years to come." ZDNet. June 2020. https://www.zdnet.com/article/ripple20-vulnerabilities-will-haunt-the-iot-landscape-for-years-to-come/The%20researchers%20said%20they%20named,and%20the%20years%20to%20come.&text=This%20vulnerability%20may%20result%20in%20remote%20code%20execution
3. A. Greenberg. "Hack brief: Microsoft warns of a 17-year-old 'wormable' bug." Wired. July 2020. https://www.wired.com/story/sigred-windows-dns-flas-wormable/
4. K. Kain, S. W. Smith, and R. Asokan, "Digital signatures and electronic documents: A cautionary tale," in *Advanced Communications and Multimedia Security: IFIP TC6/TC11 Sixth Joint Working Conference on Communications and Multimedia Security September 26–27, 2002, Portorož, Slovenia*, J.-B. Borka and K. Tomaž, Eds., Boston, MA: Springer-Verlag, 2002, pp. 293–307.
5. D. Kaminsky, M. L. Patterson, and L. Sassaman, "PKI layer cake: New collision attacks against the global X.509 infrastructure," in *Proc. Int. Conf. Financial Cryptogr. Data Security*, 2010, pp. 289–303.
6. M. Kol and S. Oberman, "Ripple20 CVE-2020-11896 RCE and CVE-2020-11898 Info Leak," JSOF, Tech. Rep., June 2020. [Online]. Available: https://www.jsof-tech.com/wp-content/uploads/2020/06/JSOF_Ripple20_Technical_Whitepaper_June20.pdf
7. M. Kol, A. Schön, and S. Oberman, "Ripple20 CVE-2020-11901," JSOF, Tech. Rep., Aug. 2020. [Online]. Available: https://www.jsof-tech.com/wp-content/uploads/2020/08/Ripple20_CVE-2020-11901-August20.pdf
8. A. Laurie and G. Wypych. "New vulnerability could put IoT devices at risk." Security Intelligence. Aug. 2020. https://securityintelligence.com/posts/new-vulnerability-could-put-iot-devices-at-risk/
9. C. Mainka, V. Mladenov, S. Rohlmann, and J. Schwenk, "Attacks bypassing the signature validation in PDF," Ruhr-Universität Bochum, Bochum, Germany, Tech. Rep., Mar. 2020. [Online]. Available: https://pdf-insecurity.org/download/report-pdf-signatures-2020-03-02.pdf
10. M. Kumar, "New flaws in qualcomm chips expose millions of android devices to hacking." https://thehackernews.com/2019/08/android-qualcomm-vulnerability.html
11. L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security applications of formal language theory," *IEEE Syst. J.*, vol. 7, no. 3, pp. 489–500, 2013. doi: 10.1109/JSYST.2012.2222000.
12. Siguza. "Psychic paper." May 2020. https://github.com/Siguza/psychicpaper
13. S. Tzadik. "SIGRed – Resolving your way into domain admin: Exploiting a 17 year-old bug in Windows DNS servers." Check Point Research. July 2020. https://research.checkpoint.com/2020/resolving-your-way-into-domain-admin-exploiting-a-17-year-old-bug-in-windows-dns-servers/
14. ZecOps Research Team. "You've Got (0-click) Mail!" Apr. 2020. https://blog.zecops.com/vulnerabilities/youve-got-0-click-mail/
15. DEFCONConference, Makkaveev S. *DEF CON Safe Mode - Slava Makkaveev - Pwn2Own Qualcomm Compute DSP for Fun and Profit*. (Aug. 5, 2020). Accessed: Feb. 13, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=CrLJ29quZY8&feature=youtu.be

Sameed Ali is a Ph.D. student in the Department of Computer Science at Dartmouth College, Hanover, New Hampshire, 03755, USA. His research interests include building and optimizing parsing algorithms for field-programmable gate arrays. He is also interested in program analysis and symbolic execution to find various input-handling vulnerabilities. Contact him at sameed.ali.gr@dartmouth.edu.

Prashant Anantharaman is a Ph.D. student in the Department of Computer Science at Dartmouth College, Hanover, New Hampshire, 03755, USA. His research interests include securing industrial control systems protocol implementations using LangSec and building verified parsers for various grammars. He is a Student Member of IEEE and a member of the IEEE Computer Society. Contact him at pa@cs.dartmouth.edu.

Zephyr Lucas is a Ph.D. student in the Department of Computer Science at Dartmouth College, Hanover, New Hampshire, 03755, USA. His research interests include exploring the computational complexity and run-time complexity of various parsing algorithms. Contact him at zephyr.s.lucas.gr@dartmouth.edu.

Sean W. Smith is a professor in the Department of Computer Science at Dartmouth College, Hanover, New Hampshire, 03755, USA. His research interests include security and privacy in software and hardware in various real-world applications. Contact him at sws@cs.dartmouth.edu.