CASTLES BUILT ON SAND

TOWARDS SECURING THE OPEN-SOURCE SOFTWARE ECOSYSTEM

ZOË BRAMMER
SILAS CUTLER
MARC ROGERS
MEGAN STIFEL

APRIL 2023

IST Institute for SECURITY + TECHNOLOGY

Castles Built on Sand:
Towards Securing the Open-Source Software Ecosystem

April 2023

Authors: Zoë Brammer, Silas Cutler, Marc Rogers, Megan Stifel
Design: Sophia Mauro
Copy Edits: Geoffrey Ballinger

IST

About the Institute for Security and Technology

As new technologies present humanity with unprecedented capabilities, they can also pose unimagined risks to global security. The Institute for Security and Technology's (IST) mission is to bridge gaps between technology and policy leaders to help solve these emerging security problems together. Uniquely situated on the West Coast with deep ties to Washington, DC, we have the access and relationships to unite the best experts, at the right time, using the most powerful mechanisms.

Our portfolio is organized across three analytical pillars: **Innovation and Catastrophic Risk**, providing deep technical and analytical expertise on technology-derived existential threats to society; **Geopolitics of Technology**, anticipating the positive and negative security effects of emerging, disruptive technologies on the international balance of power, within states, and between governments and industries; and **Future of Digital Security**, examining the systemic security risks of societal dependence on digital technologies.

IST aims to forge crucial connections across industry, civil society, and government to solve emerging security risks before they make deleterious real-world impact. By leveraging our expertise and engaging our networks, we offer a unique problem-solving approach with a proven track record.

Acknowledgments

The publication of this paper was made possible by a generous grant from Omidyar Network, a social change venture that works to reimagine critical systems and the ideas that govern them. We are grateful to Omidyar Network for their support of our research into the security of the open-source software ecosystem.

Thank you to all those who have been involved in the research for this paper, including members of the IST team, external contributors, and anonymous peer reviewers. We are grateful to all those who took the time to lend their expertise to this project.

AUTHORS

Zoë Brammer
Silas Cutler
Marc Rogers
Megan Stifel

CONTRIBUTORS

We appreciate the following contributors for their expertise and thoughtful feedback as we drafted this paper. They provided insight on technical aspects of open-source software, input on recommendations, and a diverse range of perspectives on the challenges and opportunities posed by securing the open-source software ecosystem.

John Banghart
Bryson Bort
Sven Herpig
Andrew Jensen
Jen Miller-Osborn

Table of Contents

Introduction

Software is a foundational part of the infrastructure of the modern world. While vulnerabilities can be present in all types of software, the majority of software developers rely to some extent on open-source packages to catalyze innovation in software development without rebuilding the same packages many times over. Provided that these packages are secure, open-source software creates added capacity that translates into economic gains. The impact of the Log4j software vulnerability (CVE-2021-44228), disclosed on December 9, 2021, should prompt cybersecurity professionals and the software ecosystem at large to reimagine how to mitigate open-source software vulnerabilities (For more information on the Log4j vulnerability, see Appendix 1: What Happened? and Appendix 2: How Did It Happen?).

Like Heartbleed in OpenSSL (CVE-2014-0160), remote code execution (RCE) in Apache Struts (CVE-2021-31805), and exploitation of the Bash Uploader in Codecov, Log4j is a critical vulnerability identified in open-source software (For a comparative case study of the Log4j and Heartbleed vulnerabilities, see Appendix 3: Shortening the Maturation Curve: The Log4j and Heartbleed Vulnerabilities). It is a useful case study because of the documentation of its development, the transparent response and mitigation efforts at each stage of the disclosure cycle, and its ongoing exploitation. As vulnerabilities cannot be completely eliminated and can be rapidly exploited by a wide array of actors, there is an urgent need for a plan to reduce the prevalence of vulnerabilities and to mitigate the greatest risks posed to the entire software ecosystem when they do arise—both now and in the future.

This report advocates shifting open-source software security to a shared responsibility model, redoubling support for existing secure software development frameworks, policies, and licenses, and reexamining approaches to vulnerability management and mitigation to ensure they account for open-source software (For additional context, see pages 5, 16, and 17 of Securing the Modern Economy: Transforming Cybersecurity Through Sustainability).

If adopted and implemented by stakeholders in the open-source software ecosystem, these recommendations could help reduce the impact of vulnerabilities such as Log4j and prevent future vulnerabilities from arising (Stakeholders include but are not limited to open-source contributors, organizations using open-source software, and governments

working to secure the open-source software ecosystem). Many of the recommendations echo reports published by the Cyber Safety Review Board of the Cybersecurity and Infrastructure Security Agency (CISA), the National Institute for Standards and Technology (NIST), the Atlantic Council, and others. A shared responsibility model brings such approaches together to promote a fundamental shift in the open-source software ecosystem.

The information and recommendations presented in this report are useful in different ways to different stakeholders. Readers with no knowledge of the Log4j vulnerability or open-source software should begin by reading Appendices 1 and 2 for a complete picture of the vulnerability and the challenges present in the existing open-source software ecosystem. Policy experts concerned with the security of open-source software should focus on the main body of the report and its recommendations, consulting the appendices as needed for context. Readers interested in developing a more technical understanding of open-source software vulnerabilities and how they are evolving should read Appendix 3: Shortening the Maturation Curve: The Log4j and Heartbleed Vulnerabilities. Instances where the appendices can provide useful context to the main body of the report have been flagged.

Shifting Open-Source Software Security to a Shared Responsibility Model

Open-source code, by design, is available to the general public and incorporated frequently into both commercial and open-source projects. Practices for integrating proprietary and open-source software into products differ, in part because software provided by third-party vendors usually comes with contractual obligations that open-source code does not. Internally written code usually follows an organization's secure software development life cycle (SDLC), which includes peer review. Open-source code, however, is usually integrated without a rigorous review process. Even as the broader software ecosystem defers responsibility to them, the disproportionately small—and predominantly voluntary—group of developers that maintain open-source code cannot be expected to identify and mitigate all vulnerabilities.

A shared responsibility model could distribute the responsibility of securing and maintaining open-source software more evenly. Open-source code allows commercial software developers to save time and money when developing projects (For more

information on the economics of open-source software, see Appendix 2: How Did It Happen?). Given their heavy reliance upon such software, commercial software developers should act as responsible partners in the open-source community by reporting bugs when they are found, providing support to package maintainers when possible, and avoiding shifting blame to open-source developers when vulnerabilities arise. In an ideal world, all code used by an enterprise, whether proprietary or open-source, would face the same scrutiny and validation through secure SDLC processes. Large corporations are especially well-suited for the shift to a shared responsibility model, given their resources and the benefits and risks they face when integrating open-source software into projects. Further, the federal government could develop incentives encouraging companies to adopt and leverage existing quality assurance processes to identify and rectify vulnerabilities in open-source code. Incentives could also encourage organizations to report identified vulnerabilities to the original developers, rather than simply fixing the code in their own environment.

Many businesses that already conduct audits and maintenance of proprietary code could benefit from doing the same with open-source software, applying the same standards to all code integrated into their platform. A development model that leverages open-source code to cut costs and accelerate project delivery is sustainable so long as companies test and certify all code on a regular basis. Transitioning the maintenance of open-source software to a shared responsibility model that includes an obligation to review code is the most efficient and effective way to increase the security of the entire software ecosystem.

Further, if organizations commit to reviewing and auditing open-source code on a regular basis, vulnerabilities may be caught before the code is published instead of years later. While this could introduce some risks—like the deluge of spam pull requests to open-source repositories following DigitalOcean's Hacktoberfest—or change the nature of open-source projects, leveraging existing resources is far more likely to strengthen the open-source software community than to impair it. Other mechanisms to create more secure end products should also be explored, including new roles for membership organizations, alternative contribution models and certification strategies, and funding mechanisms designed to finance maintenance and updates for established open-source projects.

Finally, stakeholders should reexamine existing frameworks for legal liability. A strategic objective of the 2023 U.S. National Cybersecurity Strategy, shifting liability around insecure software products and services is one way to spell out where responsibility lies.

The strategy notes that "too many vendors ignore best practices for secure development, ship products with insecure default configurations or known vulnerabilities, and integrate third-party software of unvetted or unknown provenance." For example, despite developers' success in quickly developing a patch for the Log4j vulnerability, commercial software using vulnerable versions continues to be identified.

Redoubling Support for Existing Secure Software Development Frameworks, Policies, and Licenses

Shifting open-source software security to a shared responsibility model would increase the practicality of existing secure software development frameworks, policies, and licenses by more evenly spreading the burden of adoption and enforcement. However, software library vulnerabilities will continue to be a reality for the open-source community; as developers write new code, new vulnerabilities will inevitably arise. As a result, it is essential that stakeholders redouble support for existing secure software development frameworks, policies, and licensing schemes to ensure that future vulnerabilities do not endanger the Internet's infrastructure. The following strategies can help stakeholders to track, monitor, and evaluate current and future uses of open-source software, which will both improve risk assessment and mitigate fallout from vulnerabilities.

Stakeholders should drive a conversation around open-source licensing—legally binding contracts between the author and the user of a software component—that focuses on identifying gaps in the common elements of the core licenses that support the open-source ecosystem. Proper licensing would provide greater consistency within the ecosystem, which would enable a wide variety of entities to adopt open-source code while reducing the risk associated with implementation for both the developers and users of open-source projects. Licensing can also increase the visibility of code changes by ensuring modifications are tracked or released back to the original project, which can limit the fragmentation that can make vulnerability management especially challenging. Likewise, there is now an opportunity to empower software bills of materials (SBOMs) through open-source licensing, which would further increase clarity during vulnerability management. Finally, licensing schemes should protect the accessibility of open-source software, a process that would

benefit from the support of lawyers and others with a deep knowledge of legal licensing frameworks.

The U.S. government should leverage federal procurement. As the largest consumer of goods and services in the global market, the U.S. government can leverage Federal Acquisition Regulations (FAR) and the Federal Risk and Authorization Management Program (FedRAMP) to increase the cybersecurity of products and services it purchases. By mandating that companies that use open-source software in what they sell to the U.S. government subject the open-source code they use to the same security processes as their own code, contribute fixes, make coordinated disclosures to the repository maintainers, avoid dead projects, and shoulder the responsibility of maintaining projects, the U.S. government can move the needle toward improved cybersecurity at scale. Additionally, ensuring that open-source software integrated into government products and services is secure could require additional support to smaller developers that might otherwise struggle to adhere to all of these requirements. By enforcing FAR and the FedRAMP, the U.S. government can also force disclosure of known flaws in products, a win for open-source developers, the companies that integrate open-source software, and the consumers that use these goods and services.

Companies and open-source developers should adopt SBOMs and thereby provide a detailed list of components used in a software product (For more information on the benefits of leveraging SBOMs to strengthen cybersecurity, see page 13 of Securing the Modern Economy: Transforming Cybersecurity Through Sustainability). It is important to note that, if not kept up to date, there can be discrepancies between SBOMs in repositories and the code that is actually running. If properly maintained, these lists can be used by enterprises and agencies alike to identify open-source components that contain particular vulnerabilities once those vulnerabilities have been disclosed. These lists can also be used proactively to understand which components represent the greatest risk and which need the most support. SBOM adoption has been slow because many companies lack the knowledge and tooling to build SBOMs, have not grasped the potential benefits of an SBOM versus the perceived effort of creating one, or require support in maturing processes. A lack of SBOM standardization further impedes adoption. However, widespread adoption would make it easier to determine the most widely used libraries, a highly useful development in terms of incident and risk management. Fortunately, the U.S. and other governments, as well as large-scale software integrators and dependents, have levers like procurement

they can use to make SBOMs essential in the marketplace. Further, CISA's efforts to uplift the value of SBOMs and help less mature organizations develop effective ones are driving improvements in their creation and adoption.

The U.S. government and private industry should support and expand initiatives like the Open-Source Technology Improvement Fund Managed Audit Program to help identify and secure open-source codebases that are widely used and rarely updated. Other proposals, such as the Sovereign Tech Fund, aim to direct funding toward improving security measures in such software. It is also worth exploring new models to assess open-source risk, as the current lack of standardized risk measurements results in largely subjective choices about which libraries to secure. Rectifying the current system should include a comprehensive strategy based on quantified risk to select which projects to prioritize.

Organizations that employ open-source components should implement quantitative risk assessment to enable effective communication around and response to vulnerabilities. Such assessments should take into account evaluations from Information Sharing and Analysis Centers and other relevant bodies like OpenSSF, scoring systems such as the Forum of Incident Response and Security Teams' Common Vulnerability Scoring System Version 3.0 (CVSS V3), as well as organization-specific context where possible. While the CVSS V3 and similar scoring systems provide a general sense of the severity of a given vulnerability, without context they do not provide enough information for an organization to adequately understand and respond to the risk that a particular vulnerability represents (For more information on the challenges posed by existing scoring systems, see Appendix 2: How Did It Happen?). As a result, companies should view CVSS V3 as a starting point for determining risk, and enrich this score with company-specific knowledge before taking action.

Developers should explore the benefits and challenges of introducing memory safety as a way to combat vulnerabilities in software. The Chromium Project found that 70% of serious bugs are memory safety problems, resulting in developers inadvertently inserting memory corruption bugs into their C and C++ code. While introducing memory safety would be resource-intensive, especially for projects that already exist, there is an opportunity for the open-source community to have a conversation about common practices that may uplift code security as a whole, potentially even eliminating entire categories of vulnerabilities.

Sector actors should incentivise more diverse skill sets in open-source software development and maintenance. Creating secure open-source software requires more than just writing code, and the sector would benefit from the addition of experts across a range of skill sets, including but not limited to cybersecurity, risk management, and project management. The inclusion of experts from a range of fields would ultimately make the development of open-source software safer and more secure. To enable this diversity, funding opportunities should be explored with an eye toward rethinking the structure of participation in open-source software maintenance and the incentives that underpin this structure.

Reexamining Approaches to Vulnerability Management and Mitigation to Ensure They Account for Open-Source Software

Even with a more proactive posture and a shift toward a shared responsibility model, some vulnerabilities will still arise. With this in mind, there are a number of ways to increase the effectiveness of reactive approaches to vulnerabilities. As evidenced by the Heartbleed and Log4j vulnerabilities, it is critical that vulnerability mitigation processes include comprehensive plans to work with governments, the private sector, and security researchers (For a comparative case study of the Log4j and Heartbleed vulnerabilities, see Appendix 3: Shortening the Maturation Curve: The Log4j and Heartbleed Vulnerabilities). These partnerships will form a stronger foundation upon which to execute reactive approaches. The following framework offers a more sustainable path to proactive vulnerability management and mitigation within a shared responsibility model.

Vulnerability management should become more closely aligned with threat intelligence through the sharing of tools and skills. Most importantly, vulnerability management will have to break traditionally slow iterative cycles and become more agile. Stakeholders understand that vulnerabilities can be developed into exploits within hours of disclosure, and it is critical that all involved parties have the necessary intelligence and fluidity to respond. It is also critical that vulnerability management efforts comply with existing risk assessment structures like scoring such that each response is appropriately prioritized. Addressing end-product security will require a better approach to patching open-source projects, including more efficient patch development and implementation.

The U.S. government, including CISA and the Office of the National Cyber Director (ONCD), should maintain threat intelligence teams that provide contextual vulnerability management assistance, especially to small and medium-sized businesses. Specifically, adapting the information collected by these teams in a way that is accessible to those at or under the cybersecurity poverty line—such as by outlining best practices and actions to avoid—would provide much-needed guidance to businesses without technical expertise (For more information on the cybersecurity poverty line, see Appendix 4: A Note on the Cybersecurity Poverty Line). Threat intelligence teams understand internal software and systems, like network security teams, and are familiar with reported vulnerabilities, like incident response (IR) teams. Additionally, they have the added capacity to proactively scan for systems that may be vulnerable and can provide context into an actor's tactics, techniques, and procedures throughout an active vulnerability instance. Many prolific forms of cybercrime, including ransomware, are evolving rapidly to exploit open-source software vulnerabilities, which are particularly attractive to affiliates leveraging simple reusable attacks given their ubiquity and long tail to remediation (For more information on long tails, see Appendix 2: How Did It Happen?).

The U.S. government should create a database of products known to contain vulnerable dependencies. For example, Workspace ONE Access Connector and VMware Identity Manager Connector are known to contain Apache Log4j. While CISA's Known Exploited Vulnerabilities Catalog tracks high-profile exploits, it does not identify products or services that may contain a vulnerability unless they experience a high-profile attack. While creating such a database would present a significant challenge, it would ultimately enable the maintenance of lists of products and versions that are a risk. Searching through VMware's entire knowledge base to check for such vulnerabilities is not a useful alternative, and the software ecosystem would benefit from a one-stop shop to identify known vulnerabilities. The database could also include configurations of software that expose potential vulnerabilities. This would help mitigate the risk posed by companies that claim their products are not vulnerable when in reality a simple configuration change could expose the vulnerable code to exploitation.

Conclusion

The software ecosystem has an opportunity to improve current approaches to addressing open-source software vulnerabilities. In managing software risks, organizations generally rely on network security teams to identify known vulnerabilities within their networks (For more information on approaches to vulnerability management, see Appendix 2: How Did It Happen?). In cases where defenders identify evidence of a vulnerability being exploited, these teams work alongside IR teams and other consultants.

This reactive model only works in instances of a known threat where mitigation efforts can outpace threat actors. In the case of the Log4j vulnerability, many companies do not know if they have Log4j in their products; the only way it can be found is to open up the code. To make matters worse, Log4j has a long tail, meaning it will keep popping up in products and services for years to come. This risk management model is inefficient and unsustainable, as it requires responders and other consultants to maintain around-the-clock defense. Given the rapidly shifting ecosystem, it is becoming evident that a cultural shift in the cybersecurity of open-source software is necessary.

Rather than a reactive approach, the software development ecosystem must shift code review to an earlier stage in the development and deployment lifecycle. This report advocates for shifting open-source software security to a shared responsibility model, redoubling support for existing secure software development frameworks, policies, and licenses, and reexamining approaches to vulnerability management and mitigation to ensure they account for open-source software. These changes will reduce the pressure on the open-source developers who build and maintain the software underpinning a vast array of goods and services.

There is a role for everyone to play in securing the open-source ecosystem. Governments should incentivise companies that employ open-source code to commit resources to code maintenance, support existing secure software development frameworks and policies, and leverage federal procurement to increase baseline cybersecurity in open-source products. Further, the U.S. government should examine ways to maintain threat intelligence teams that provide contextual vulnerability management assistance through agencies like CISA and NIST in collaboration with offices such as the ONCD.

Open-source developers deserve more than applause for the energy, time, and resources they dedicate, often uncompensated, to the software ecosystem. Developers and other key software stakeholders should explore pathways to incentivize the implementation of memory safety to combat potential vulnerabilities in products, and adhere to SBOMs and open-source licensing schemes.

Finally, companies employing open-source software should dedicate resources to maintaining open-source projects; develop and enforce quantitative risk assessments that take organizational context into account; and not rely entirely on CVSS V3 base scores, which offer an incomplete assessment of risk.

Appendices

Appendix 1: What Happened?

Timeline

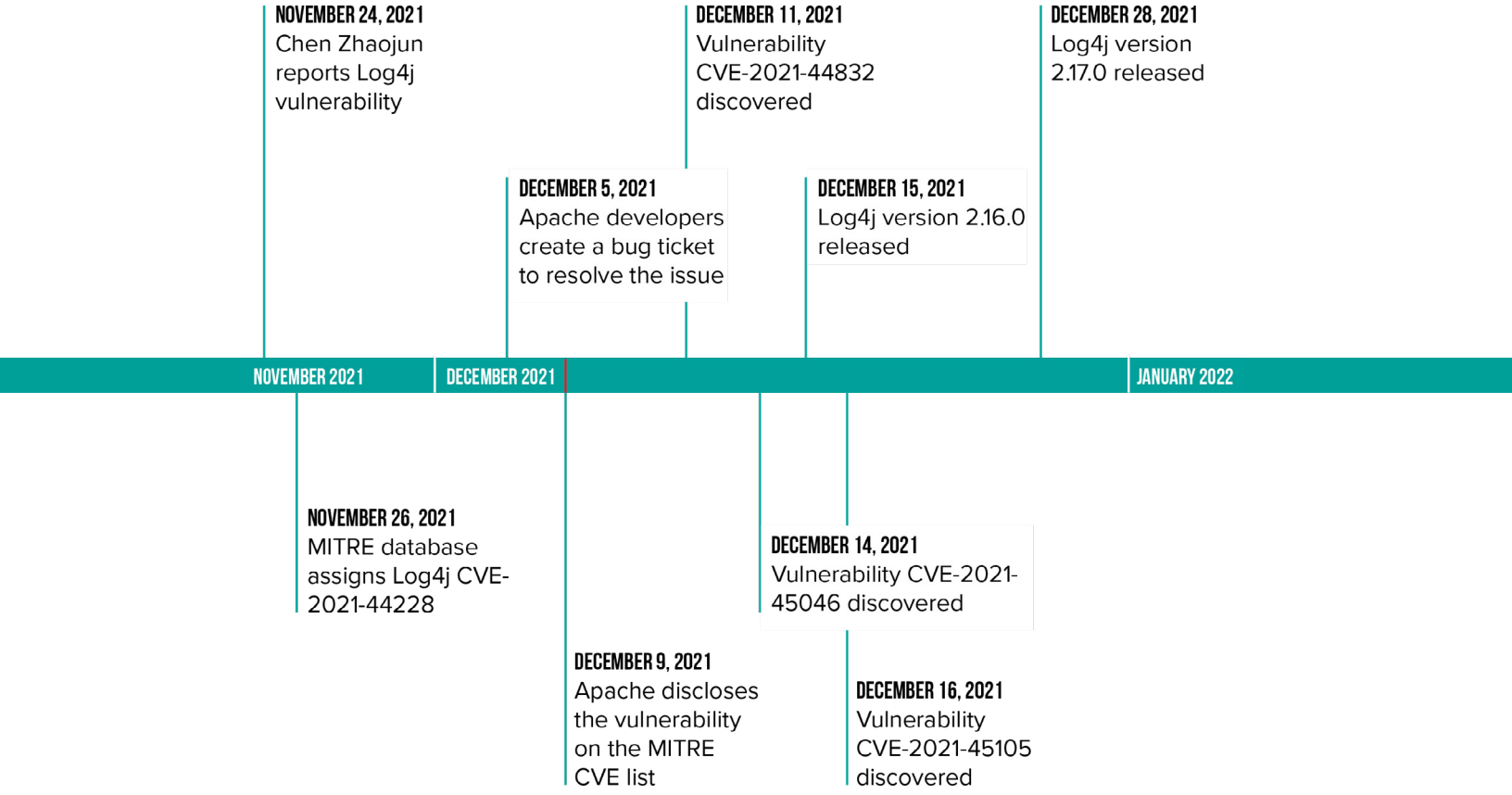
On December 9, 2021, the MITRE Corporation publicly designated a Common Vulnerabilities and Exposure (CVE) identifier to a critical security vulnerability called Log4Shell (CVE-2021-4428) housed in Log4j, a widely used open-source logging software that tracks how people interact with software platforms and applications (The MITRE Corporation maintains the CVE database, which is used to track known cybersecurity vulnerabilities). Because so many developers use Log4j as a foundational piece of their products and services, the resulting risk from this vulnerability posed to applications across the Internet was profound. As a result, there was significant concern about the reach and the implications the vulnerability would have both immediately and over time.

In the hours that followed the public disclosure, cybersecurity teams, volunteer researchers, and response firms dropped everything to begin an around-the-clock campaign to mitigate the vulnerability. As they worked, it became increasingly clear that Log4j's flaw posed a profound threat to the integrity of the Internet itself. A unique set of factors created the danger of the vulnerability, namely the capacity it afforded malicious actors to perform unauthenticated RCE in a commonly used, easily overlooked

component. Although early reports claimed that attackers were not using the exploit to attack critical infrastructure, there is now evidence that North Korea's Lazarus group is using the vulnerability to attack U.S. energy companies. In short, software, servers, and machines across the globe that utilize unpatched versions of Log4j are vulnerable to exploitation. Log4j illustrates the significant cybersecurity threat endemic to the current approach to open-source software.

Within hours of the public announcement, a security researcher dropped a proof-of-concept (POC) exploit for Log4j on Twitter, broadly enabling both legitimate network defenders and malign actors to leverage this exploit.

Log4j Vulnerability Identification Timeline



Security researchers played a critical role in the days following the public identification of the vulnerability. The wide-scale sharing of relatively simple POC scripts enabled both network defenders and threat actors to leverage the accelerated exploit development. However, it is important to note that this development largely benefited defenders, as rapid exploit development by threat actors typically occurs behind closed doors. For malicious actors, successful exploitation led to unauthenticated RCE, providing access to servers in organizations across the world and allowing them to deploy malware and

maintain access to compromised networks. Security researchers also identified affected software and hardware products, and began helping to take down attackers and notifying vulnerable parties.

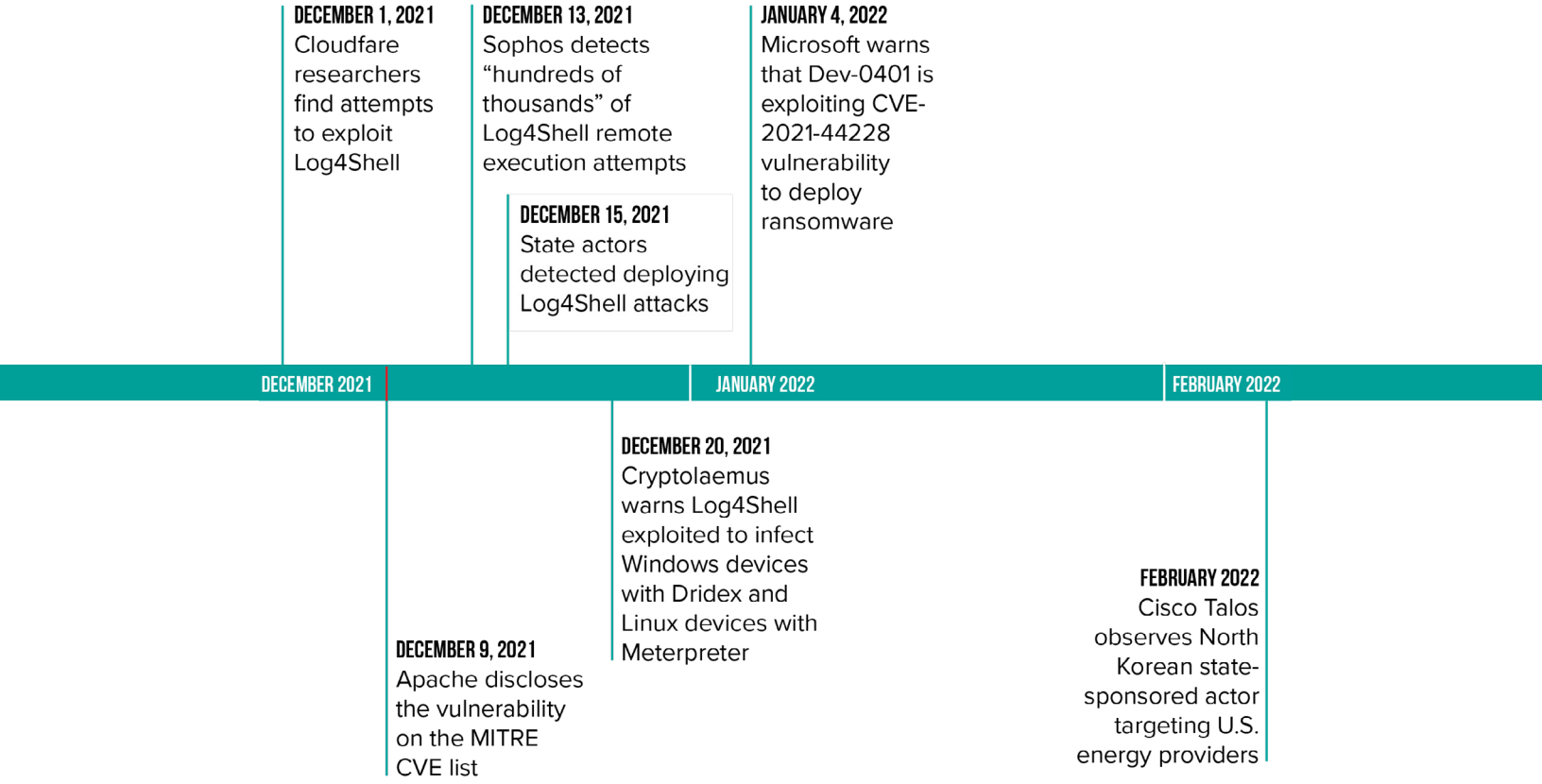
Ultimately, CISA deemed the Log4j vulnerability so serious that it issued an emergency directive, while the Federal Trade Commission issued a relatively rare warning instructing companies to remediate the vulnerability. In January 2022, the White House convened an open-source security summit seeking collaboration from industry giants to understand the mechanisms that could prevent this type of vulnerability from happening again. Further, in February, the U.S. Senate Committee on Homeland Security and Governmental Affairs called a hearing around Log4j, inviting the president of the Apache Software Foundation as well as several private sector and nonprofit experts to testify about the context surrounding Log4j and its associated vulnerability.

With the addition of the Log4j vulnerability by Apache to MITRE's CVE database and the National Vulnerability Database, MITRE acknowledged that the vulnerable Java Naming and Directory Interface (JNDI) that Log4j uses was committed to the original codebase in July 2013, over eight years before the vulnerability was publicly disclosed.

The time between the introduction of the code and delayed discovery of the vulnerability highlights a problem with vulnerability management: it largely functions as a reactive business practice of quashing vulnerabilities when they arise, instead of proactively studying and testing ubiquitous open-source libraries for dangerous vulnerabilities before they can be exploited. During the eight-year period in which Log4j included this vulnerable code, it would have been possible for a threat actor to have identified and selectively leveraged this vulnerability as part of their operations.

Exploitation

Log4j Exploitation Timeline

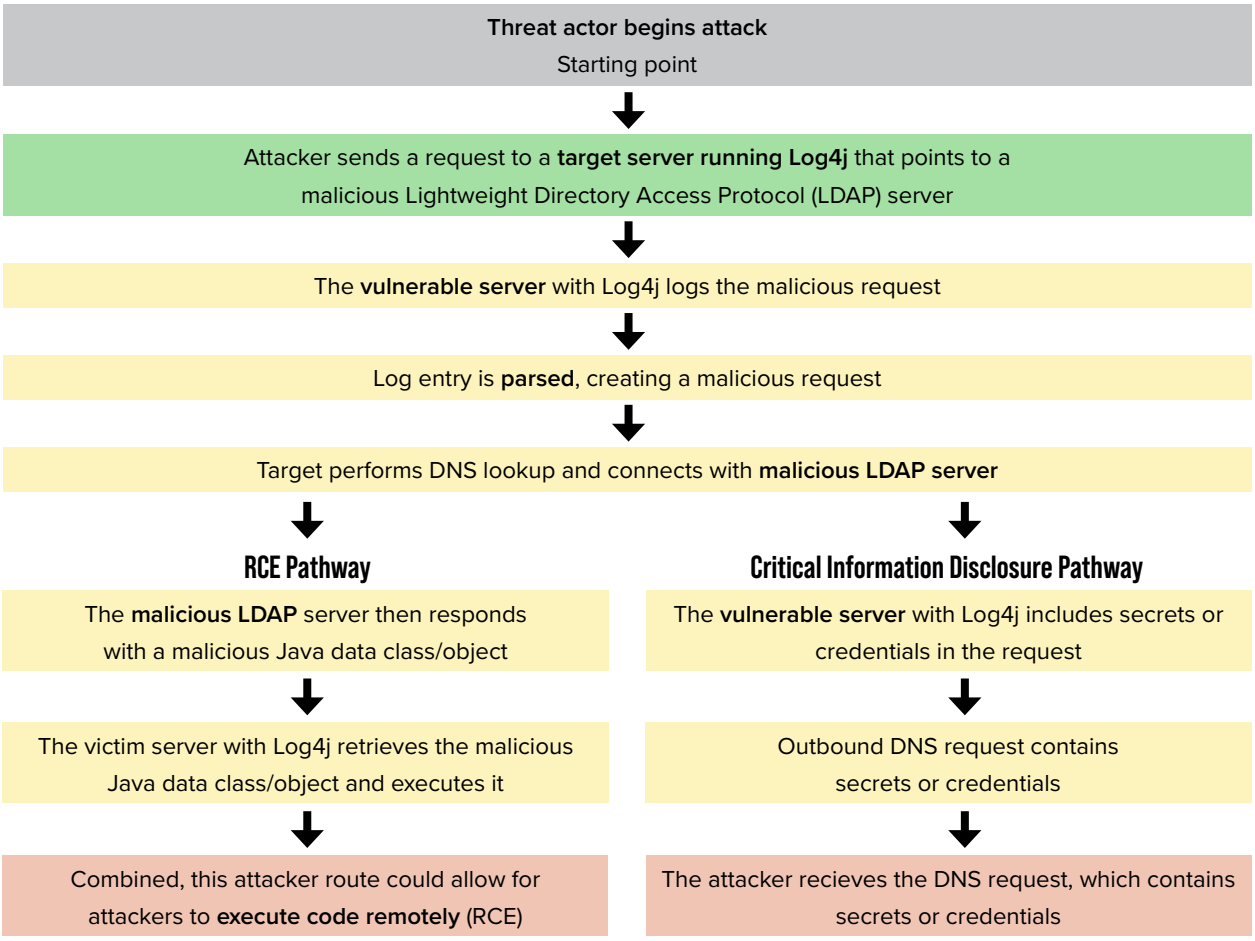


The breadth of Log4Shell exploitation offers a useful case study for the future management of vulnerabilities and their associated threats. Most commonly, threat actors exploit Log4j using JNDI lookups, according to security researcher Sophos, as well as previously unknown pathways such as web forms. Thus, threat actors could leverage the vulnerable JNDI lookup feature to inject code into servers with remote-enabled services, allowing access from a geographical distance through network connections.

The exploitation pathway changed rapidly in the weeks following public disclosure of Log4Shell, with actors evolving the most prevalent payloads—or malware intended for the victim—to bypass detection and protection methods like web application firewall (WAF) rules. Log4Shell is a relatively unique case in malicious code detection because threat actors were able to adapt payloads to avoid new WAF rules almost immediately, keeping threat responders on the defensive throughout the beginning of the exploitation.

The secondary path of exploitation came in the form of information theft using the domain name system (DNS) protocol as a covert channel attack pathway. While the individual vulnerability is relatively difficult to leverage into a usable RCE, information theft is low-hanging fruit, which makes Log4Shell problematic for the organizations that use the Log4j library by allowing actors to use the exploit to retrieve information from compromised servers or applications.

Primary Attack Pathways for Log4j Vulnerability



Using Log4Shell to Deliver Malware

In the days after the Log4j vulnerability disclosure, cybersecurity researchers at Sophos detected hundreds of thousands of attempts to remotely execute code on a wide array of industry targets, including cloud service providers, virtual service providers, and healthcare organizations. These reports identified attackers attempting to leverage the vulnerability to install cryptocurrency-mining malware, as well as several botnets including Mirai, Tsunami, and Kinsing attempting to exploit Log4j to exponentially

increase their number of nodes. Further, according to reports from Microsoft, ransomware strains like Nemesis Kitten and Khonsari launched attacks against enterprise infrastructure, while ransomware operators like Lockbit and nation-state hackers worked to develop sophisticated ways to take advantage of the new attack vector. Researchers at Microsoft also warned about attempts to install Cobalt Strike on vulnerable systems, a strategy that could allow attackers to steal usernames and passwords from a host of organizations. Even more concerning, a report from AdvIntel indicates that the Conti ransomware group used the Log4j vulnerability to gain access to some VMware servers, allowing Conti to encrypt some virtual machines. The Log4j vulnerability is a unique case because exploits did not evolve naturally over time; instead, multiple pathways were explored simultaneously in what appeared to be a race to see which could be leveraged first.

Early mass exploitation of the Log4j vulnerability saw the deployment of commodity distributed denial of service malware (recent news report from BleepingComputer about a new botnet that targets Linux-based systems and uses the Log4j vulnerability to infect new hosts) and crypto-mining campaigns (detailed report on actors that used the Log4j vulnerability to install crypto-mining malware on computers from Darktrace, an IT company that focuses on cyberdefense). While the exploit was initially more difficult to leverage in a sophisticated way that enabled RCE, the more readily available pathway involves triggering the exploitation via JNDI lookups for data exfiltration from servers that use the Log4j library. Using this lookup feature allows an actor to probe the server for important information, in one instance allowing threat actors to see information related to Amazon Web Services (AWS) secret access keys that could have led to AWS instances being taken over, according to cloud service company Akamai. Data exfiltration leveraging the Log4j vulnerability is difficult to prevent because it would require shutting off a server's contact with DNS, an infeasible solution.

Barriers to Exploitation

While the Log4Shell exploit may be easy to trigger, using it to successfully inject malware requires malicious actors to exploit the Log4j library through an asynchronous process where some functions run "in the background," adding complexity to the attack process. The actor uses a JNDI lookup to push malicious code into a server, but that code needs to be executed to actually work, which can be difficult in an asynchronous environment. Egress rules—the rules that specify what kind of information can enter and exit an application—vastly complicate efforts to retrieve usable output from JNDI RCE. Additionally, successful exploitation of an RCE requires threat actors attempting

to leverage Log4Shell to estimate where the RCE would be effective in the architecture of an application. A malicious JNDI lookup through Log4j is not inherently successful or effective. This means that **the highest-risk applications are those that contain code that has a known, reliable exploitation pathway located in an architecture that the attacker understands.**

The Politics of Vulnerability Disclosure

That advanced persistent threats (APTs) are well-resourced to utilize Log4Shell points to a complicating factor of open-source software security: the politics of vulnerability disclosure. The political undercurrent of the Log4j disclosure is a bellwether of the future of vulnerability management—a process marred by increasing globalization and exponential interconnectedness among interdependent applications. As this future is realized, leaks that allow malicious actors to compromise vulnerabilities before a fix can be deployed threaten to radically change the landscape of threat hunting and mitigation and the systems they underpin.

New regulations around zero-day vulnerabilities passed by the Chinese government in July 2021 contextualized and complicated the Log4j vulnerability disclosure. The new regulations instruct Chinese citizens who identify new vulnerabilities to "tell the [Chinese] government, which will decide what repairs to make," according to reports from the Associated Press. In this case, upon public release of the exploit, the Log4j team noted that Chen Zhaojun, a security researcher from the Alibaba Cloud Security Team, reported the vulnerability on November 24, 2021, two weeks before the public disclosure—despite the recent change to Chinese disclosure legislation.

China's Ministry of Industry and Information Technology swiftly condemned the disclosure and instructed the Cyberspace Administration of China to suspend its information-sharing agreement with Alibaba Cloud for six months, citing the Log4Shell disclosure as the reason for suspending the partnership. The Chinese government also levied sanctions against Alibaba. The suspension was an arbitrary punishment for Alibaba's responsible disclosure, designed to embarrass the company and dissuade others from sharing software vulnerabilities, creating a sort of chilling effect around vulnerability sharing. The fact that researchers responsibly disclosed the vulnerability to the Apache Software Foundation in the case of Log4j was a result not of international norms and standards, but a sense of moral responsibility on the part of developers at Alibaba. In this way, the discovery and disclosure of a vulnerability in Log4j was an instance of personal judgment

rather than the result of a standard process, further underscoring the existential threat posed by the current approach to vulnerability management.

On November 26, two days after Chen Zhaojun reported the original Log4j vulnerability, the MITRE database assigned it the identifier CVE-2021-44228. Researchers at Cloudflare found that attempts to exploit the vulnerable code started as early as December 1, 2021, eight days prior to the public announcement of the vulnerability.

Appendix 2: How Did It Happen?

What is Open-Source Software?

Open-source software libraries, packages, and modules present batches of prewritten code that can be imported into programs or applications to speed up the software development process. Given the scale of open-source use cases, this type of software arguably forms the foundation for the development of the future Internet, in addition to serving as the basis for much of today's Internet. Software programming libraries—sets of prewritten code that performs certain tasks—like Log4j exist across hundreds of programming languages to aid with data configuration, documentation, message templates, subroutines, classes, values, and type specifications. Libraries allow for the ability to reuse a behavior, such that a program gains the behavior implemented inside that library without having to implement the behavior itself. In essence, programming libraries provide shortcuts that save stakeholders time and money in software and program development.

Log4j is an open-source library that functions as a logging framework for software applications. Logging is a critical feature for applications because it allows developers to understand how both software and users are interacting with a given software platform. Logging registers information about the various events, use cases, and errors that occur within an application. As a result, developers can effectively debug software applications by understanding the conditions that lead to a bug. In Senate testimony, Apache Software Foundation President David Nalley stated that Log4j records operating events within application and storage management software, software development tools, virtualization software, and some video games. The Log4j library is one of the most popular logging tools for Java applications and has a wide range of uses, from logging

application behavior to collecting information for business analytics. Java is an extremely popular programming language because it can be used across many architectures.

The number of usable libraries has exploded with the continuing development of the Internet. Today, for example, there are almost 2 million libraries for the node.js programming language, a number that grows by nearly 1,000 daily. A single vulnerability in any of these libraries has the potential to cause massive security issues for dependent applications. The infrastructure and ubiquitous use of open-source software means that cybersecurity operations will likely feel the impact of vulnerabilities such as Log4j for years to come. While many observers of the crisis point out Log4j's long tail, beyond patching, we have yet to see a comprehensive plan for how to root out this vulnerability from the products and services it inhabits.

Between August and November 2021, the Log4j library was downloaded 28.6 million times, placing it in the top 0.003% percentile of popular libraries, according to research from Sonatype. **In the weeks after the Log4j disclosure—despite the efforts of the Apache Software Foundation to point users toward the updated Log4j library—nearly 65% of downloaded versions were the vulnerable version of Log4j. Four months after Apache disclosed the Log4j vulnerability to MITRE, security firm Rezilion found that 36% of the Log4j versions actively downloaded from Maven Central, a public repository of Java libraries, were still vulnerable to the Log4Shell exploit, highlighting how difficult it is to get developers to update their software packages.** Further, Rezilion identified over 90,000 public web servers that were vulnerable to the Log4Shell exploit.

There are over 500,000 active repositories that depend on the original Log4j library, according to the Log4j dependency graph on GitHub. In addition, according to Google Security, the Log4j vulnerability impacted more than 35,000 Java packages, representing over 8% of the most significant Java package repository. The vulnerability in this single library impacted companies like Apple, Microsoft, IBM, Siemens, and Salesforce, as well as service providers like N-able and ConnectWise. This is likely only the tip of the iceberg. From the Internet of Things to automotive use components, many embedded systems include Log4j, and there is no way to know without directly examining the code to verify the versions used.

Because programming libraries provide shortcuts in software and program development, even billion-dollar software companies use open-source libraries to build software. Given how widespread open-source infrastructure is today, vulnerabilities like Log4j take much longer to fully remediate, a concept known as a "long tail." Though the damage may not be immediately severe, addressing the Log4j vulnerability is like removing asbestos

from an old building: the vulnerability is just as pervasive, difficult to locate, and capable of causing problems well into the future. In addition, the vulnerability is often well-hidden behind complex systems that are not exposed to the Internet. Remediating the vulnerability necessitates cooperation from the network owners, who must lower some protections and open the system to outside parties. Each iteration of the code utilizing Log4j requires individual patching, which means that millions of apps will need to be individually patched—a process that will likely take years to fully realize.

Aside from saving development time, the most important impact of open-source software is that it allows software development companies to maintain their position at the forefront of innovation. Open-source software is one of the primary factors behind the explosion of cutting-edge software development in recent decades, with developers able to focus on creating new software rather than rebuilding the same packages many times over. This added capacity has been a key source of economic development and innovation in the twenty-first century, highlighting the importance of maintaining and securing open-source libraries. As organizations embrace software development and digital transformation, open-source software will become increasingly foundational to the Internet and next-generation software development.

Finally, open-source software is often critical to the infrastructure of software development companies. Linux, an open-source operating system, runs all of the world's 500 supercomputers and 23 of the top 25 websites in the world, according to reporting by 99firms. High-level computing, software development, and website development would all be much more difficult and time-consuming without open-source software. Given its importance to development, open-source software has a significant impact on the ecosystem as a whole.

The Economics of Open-Source Software

The economic impact of open-source software is difficult to gauge because organizations generally use the software without returning any metrics around value or usage back to the original developers. In the UK, for example, open-source software contributes $59 billion each year to GDP and provides an estimated $63 billion in potential value for UK businesses, according to a study from OpenUK. The lack of shared data about usage and integration from organizations that utilize open-source software is an important signal of a structural issue prevalent in its use. Although many organizations use open-source software, few provide data or feedback in return for the functionality provided.

Software giants widely utilize open-source software to build monetized platforms, technologies, and services, but do not always invest those cost-savings in supporting the foundational software upon which they build these profitable services. The Linux foundation, for example, estimates that open-source software constitutes 70-90% of any given piece of modern software solutions. These companies use open-source software through a commercial license, which requires very little accountability. Companies integrate open-source software into their own software in unique and novel ways, which can open the door for new vulnerabilities, especially if those organizations do not share nonstandard configurations with the original software foundation.

The unbalanced relationship between organizations and open-source software projects exists because companies that utilize this software embrace business models that rely on concepts like lean startup and agile development, and turn to open-source libraries to shorten their development curve. Without proper systems in place to track and maintain codebases and their deployment, vulnerabilities within open-source and third-party software libraries can cause cascading failures throughout the Internet, even if they usually do not.

What Happens When Things Go Wrong?

To date, responding to critical vulnerabilities has been a reactive process, with IR teams working on an emergency basis. This posture has been effective against vulnerabilities like Heartbleed, in which there was one pathway for exploitation. However, issues like Log4j offer multiple pathways for exploitation, which presents a challenge because each new pathway may require dedicated engineering time to develop detection countermeasures. Log4j underscores the criticality of vulnerabilities with rapid maturation and a long tail. As new vulnerabilities arise and mature at an increasingly rapid pace, relying on a reactive model is inefficient (For more information on rapid maturation, see Appendix 1: What Happened?). Log4j has highlighted the pressing need to rethink the current model, which includes examining the way that incident responders prioritize and respond to vulnerabilities.

Reactive models tend to be driven by compliance and static measures like risk scoring systems that determine how and when IT and IR teams respond to incidents. Scoring systems lack crucial context and are regularly altered, resulting in workflows that do not match the urgency and severity of threats (see Reexamining Approaches to Vulnerability Management and Mitigation). For example, Log4j V2.15 was identifiably vulnerable, but its risk score was initially set at three (defined as to be repaired in several weeks).

After outcry from security experts, the score increased to nine (to be repaired in hours). The vulnerability in Log4j V2.15 requires complex preconditions, which means that the vulnerability poses different threats to different organizations depending on their relative exposure. Given that the vulnerability score varied widely in assessments over time (from three to nine) some organizations diverted resources to deal with a vulnerability that may not have impacted them. This could be in part because accurate risk scoring exists only in the context of organizational instantiation.

These static measures similarly complicated the response to a denial of service vulnerability found in Log4j V2.16, which received a score of seven. Similar to Log4j V2.15, V2.16 is vulnerable only in nonstandard configuration. Though compliance dictated that all organizations needed to fix the vulnerability in seven days, the vulnerability only impacted organizations that adjusted the default parameters of Log4j. The result was the same: static measures of vulnerability, severity, and urgency did not take into account the context and nuance surrounding the vulnerability, because any final score is meant to be created at the organizational—or even system—level. That this does not happen routinely results in wasted time, energy, and resources.

The reactive model has several other drawbacks. First, this model does not take into account the idea that vulnerabilities can be chained together. Low-risk vulnerabilities that have a low score can be chained together to create a higher-risk vulnerability. One example of such an exploit is the Hot Potato Windows Privilege Escalation, which achieves a man-in-the-middle attack. In this escalation, a hacker will try to log into a victim's server, and when the server asks for a password the hacker gives a reply that catches the hash sent by the server. This hash can then be cracked—a potentially time-intensive process—or leveraged as part of a pass-the-hash attack to gain access to target systems. Taken independently, these vulnerabilities might each be of low or medium severity, but become much more dangerous when combined. This is another reminder that the current scoring systems and the ways they are employed often miss important context.

Context is key to responding to vulnerabilities, such that successful responses require a more complex system of vulnerability management, rather than the use of IR systems that are driven by compliance. A more proactive system could include creating and managing threat intelligence teams that provide contextual assistance for vulnerability management. Threat intelligence teams understand the internal software and systems, like IT teams, and are familiar with reported vulnerabilities, like IR teams. Additionally, however, they have the capacity to proactively scan for vulnerable systems and can

provide context into an actor's tactics, techniques, and procedures during an active vulnerability exploitation.

The technical shortcomings of the reactive model are evident, especially in response to exploits like Log4j that mature, become weaponized, and are deployed by hostile actors in a matter of hours. Further complicating this issue is the difficulty faced by small development teams in maintaining open-source resources. When using open-source code, a developer generally selects code that will be useful for their project and adds it. This means open-source code projects lack a proper auditing process and often carry redundant code that is completely unnecessary and potentially a security risk.

The Log4j example reflects this, especially with regard to the library's low percentage of functionality actually used when incorporated into projects. The vast majority of cases that integrate Log4j into a platform do not use the JNDI lookup feature that carries the Log4Shell vulnerability, which was added as an edge case function in 2013 and then largely forgotten. This kind of redundancy translates rapidly into increased risk.

This kind of risk is extremely difficult to combat because it requires that libraries be tested, inspected, and updated on a regular basis. Despite the massive scale of implementation, twelve full-time employees and two contributors maintain the Log4j library. Addressing open-source software vulnerabilities requires reassessing the way software developers maintain open-source code.

If the maturation curve is becoming shorter for all actors, the static scoring system fails to account for context, and the open-source development teams tasked with maintaining the code are under-resourced, a reactive model of incident and vulnerability management may be too risky for some organizations. It is time for a new model.

Appendix 3: Shortening the Maturation Curve: The Log4j and Heartbleed Vulnerabilities

As highly sophisticated groups begin to adapt and iterate exploits more quickly, it is critical to evolve to match the changing vulnerability ecosystem. Log4j is not the first open-source library to have a massive and misunderstood vulnerability. In 2014, members of the Google security team discovered the Heartbleed vulnerability, a security bug in the OpenSSL cryptography library used to manage many of the cryptographic algorithms deployed in various Internet security standards. When Apache first disclosed Heartbleed to MITRE, estimates claimed that nearly half a million web servers were vulnerable, causing panic similar to that surrounding the Log4j vulnerability announcement. The initial chaos eventually subsided, but research from Shodan found that in 2017—five years after Heartbleed's discovery—nearly 200,000 servers were still vulnerable around the world. Though some were likely temporary websites set up by web developers, there are thousands of active sites across the Internet that may still be exposed to the Heartbleed vulnerability.

The comparison is especially enlightening, however, in the ways Heartbleed differs from the Log4j vulnerability. Exploitation of Heartbleed has not been as prolific as Log4j in large part because the Heartbleed exploit matured slowly, with malicious actors taking longer to weaponize iterations, allowing security response teams to keep pace. In short, Heartbleed had a shallower maturation curve.

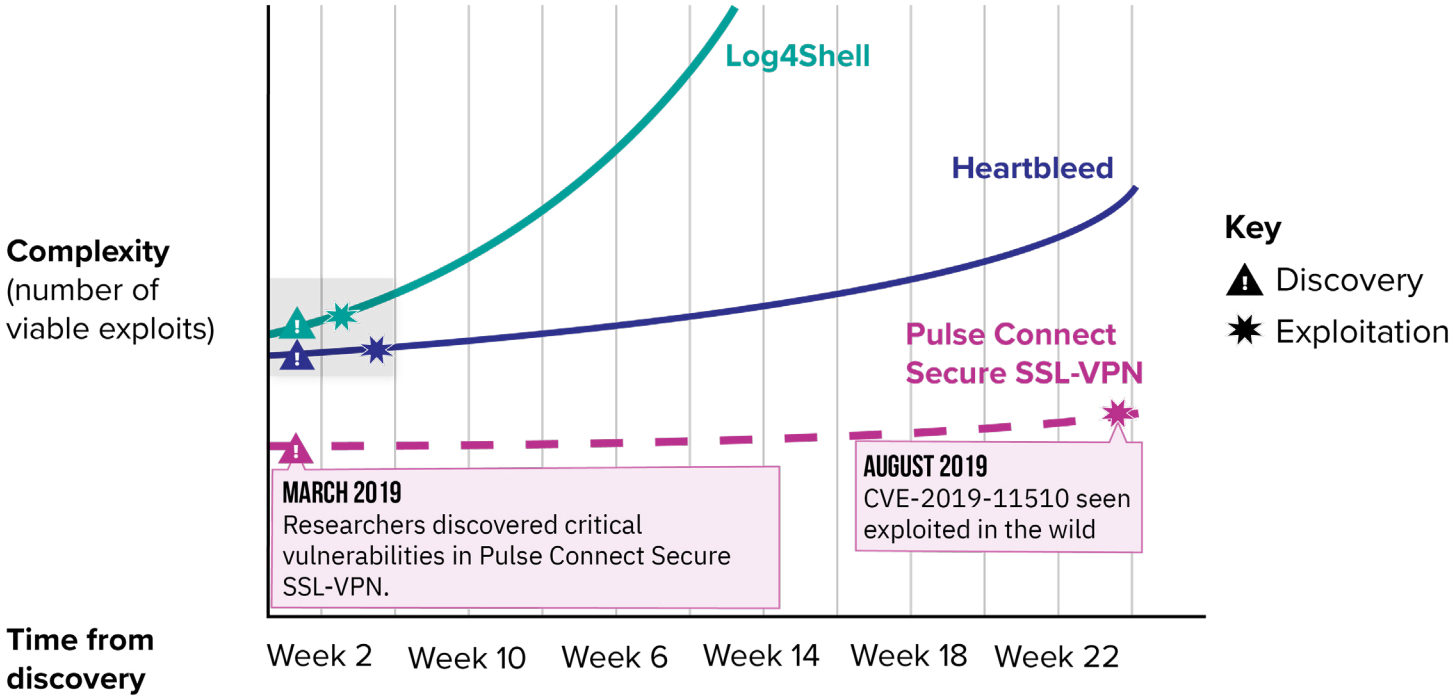
Log4j exploitation, by contrast, has evolved quickly, with the proliferation of cybercrime groups dynamically altering the vulnerability. Log4j went from a general vulnerability to a highly complex vulnerability with multiple exploitation pathways in the span of a single week. A large-scale effort by threat actors enabled early exploitation of the vulnerability. At that stage, malicious actors were essentially "throwing stuff at the wall to see what sticks." Within twenty-four hours of the disclosure, cryptominers and botnets had moved in, increasing the maturation of the exploit and allowing more malicious and larger-scale actors to move in.

To avoid wasting resources, ransomware as a service providers with affiliate groups prefer mature, predictable exploits that are tried and tested. As a result, Log4j went from a general exploit to a highly complex vulnerability leveraged by ransomware groups and other, more prolific, cybercriminal groups in a single week, with threat actors like Conti, Lockbit, and Nemesis Kitty weaponizing Log4j for ransomware.

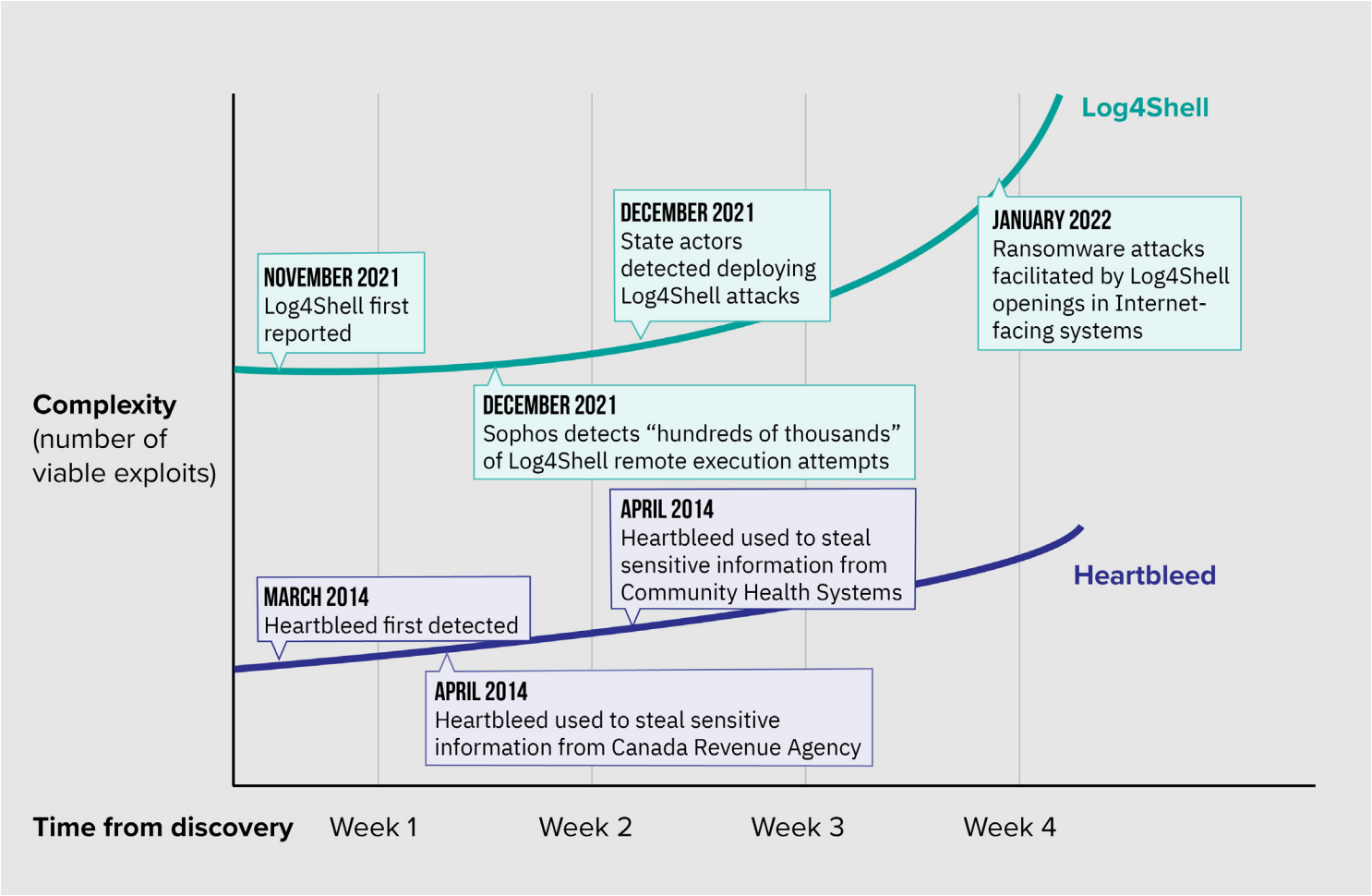
Lastly, APTs started probing the web for Log4j vulnerability indicators. Microsoft's threat intelligence team identified nation-state APT hacking teams from China, Iran, North Korea, and Turkey as adversaries exploiting Log4j. In the months since the vulnerability's discovery, for example, Cisco Talos reported that it observed Lazarus, a North Korean hacking group, targeting unnamed energy providers in the United States, Canada, and Japan between February and July 2022. The hackers used Log4Shell to compromise Internet-exposed VMware Horizon servers and establish an initial foothold in a victim's enterprise network, before deploying bespoke malware known as "VSingle" and "YamaBot" to establish long-term persistent access.

APTs are harder to track and have far more resources than cybercrime groups, and can therefore steepen the maturation curve faster than low-level cybercriminal actors. This presents novel challenges to cybersecurity professionals, who must manage the response to large-scale, low-complexity attacks and lower-scale, high-complexity attacks.

Vulnerability Maturation Curves: Log4Shell and Heartbleed vs. Pulse Connect Secure



Zooming in on Log4Shell and Heartbleed



As depicted in the above graph Zooming in on Log4Shell and Heartbleed, in the decade since Heartbleed, the maturation curve for exploits of its kind has increased rapidly. More and more groups are able to steepen the exploit maturation curve using novel techniques that threaten to make exploits more complicated and harder to defend against. As seen in the graph Vulnerability Maturation Curves: Log4Shell and Heartbleed vs. Pulse Connect Secure, the heavily exploited vulnerability CVE-2019-11510, a critical unauthenticated RCE vulnerability in an SSL-VPN product, illustrates a more traditional exploit development process, where the patch and exploitation development cycle is long enough that patching per current compliance guidelines can still be effective in shoring up vulnerabilities. Together, these examples illustrate the rapid change in exploit maturation over time. This emphasizes the need for increased focus intelligence and risk driven vulnerability management rather than set time intervals.

Appendix 4: A Note on the Cybersecurity Poverty Line

The term "cybersecurity poverty line" refers to organizations that lack the budget and/or resources to be able to effectively implement the cybersecurity measures they need. Many businesses, especially small- and medium-sized ones, do not have full-scale cybersecurity teams, and the vast majority of all businesses do not have threat intelligence teams. Both kinds of in-house expertise would provide more proactive cybersecurity. Instead, most businesses maintain a small group of network security professionals who spend much of their time setting up and changing configurations, rather than hunting for the next threat. These network security teams largely exist to defend an organization's network and essential infrastructure. Threat intelligence is a more specialized field that requires putting vulnerabilities into context and hunting for the next instance of a disastrous vulnerability, effectively putting organizations on the offensive in terms of vulnerability management, rather than on the defensive.

One of the best arguments for a cultural shift away from defensive vulnerability management is that exploits like Log4j have rapidly accelerating maturation curves (For more information on maturation curves, see Appendix 3: Shortening the Maturation Curve: The Log4j and Heartbleed Vulnerabilities). As described above, actors did not immediately exploit Log4j to deploy ransomware in all the companies that use the library, because exploits require time, energy, and resources from cybercrime groups to mature. As the maturation curve steepens, hostile actors weaponize and deploy exploits in a matter of hours. If the maturation curve becomes shorter and steeper for all actors and static scoring systems do not account for organizational context, a reactive model of incident and vulnerability management becomes too risky. In such a situation, organizations are forced to operate against the clock to remediate difficult vulnerabilities, rather than spending time beefing up infrastructure against the next attack.

INSTITUTE FOR SECURITY AND TECHNOLOGY

www.securityandtechnology.org

info@securityandtechnology.org