Kommentar zu Log4j: Es funktioniert wie spezifiziert

Originalartikel

Backup

<html> <header class=„article-header“><h1 class=„articleheading“>Kommentar zu Log4j: Es funktioniert wie spezifiziert</h1> Kristian Köhntopp </header><figure class=„aufmacherbild“><figcaption class=„akwa-caption“>(Bild: Victor Moussa/Shutterstock.com)</figcaption></figure> Über den Java-Slogan „Write Once, Run Everywhere“ wurden schon viele Witze gemacht. Den log4j-Exploit behandeln viele nun wie einen Bug – aber das ist er nicht. Eine kritische Lücke in der Java-Bibliothek Log4j beherrscht gerade die Schlagzeilen. Die IT-Welt ruft „Warnstufe Rot“ aus – weil offenbar der log4j-Code JNDI-Variablenexpansion vornehmen kann. <header class=„a-boxheader“ data-collapse-trigger=„“>Ein Kommentar von Kristian Köhntopp</header><div class=„a-boxtarget a-boxcontent a-inline-textboxcontent a-inline-textboxcontent–horizontal-layout“ data-collapse-target=„“><figure class=„a-inline-textboximage-container“></figure> Kristian Köhntopp hat 1983 angefangen zu programmieren, und ist seit 1988 online. Seitdem hat er verschiedene Dinge mit Computern aller Art getan, denkt angesichts der Lage aber über eine Karriere als Landschaftsgärtner nach. Doch was ist JNDI? Jindi al Dap ist der Name eines alten arabischen Philosophen und Mathematik-Pioniers, der für Sun/Oracle gearbeitet hat, um ein System von Directory Lookups in Java [1] zu entwickeln. Dieses System lädt irgendwie Code aus dem Internet nach. Aber selbst, wenn man länger auf das Systemdiagramm starrt, erkennt man nicht unbedingt sofort, an welcher Stelle sich der Java CLASSPATH so erweitert, dass er das gesamte Internet umfasst: <figure class=„a-inline-image a-u-inline“> </figure><h3 class=„subheading“ id=„nav_nichts_ist0“>Nichts ist jemals simpel</h3> Das ist so, weil in Java nichts jemals simpel ist. Java Code ist unstrukturierter trockener Staub von Codefragmenten in Klassendateien, die inert in keiner Weise miteinander interagieren. Erst mit den passenden Factories, Delegates, Generators und ClassLoaders werden sie instanziiert und zusammengesetzt. Der entstehende Haufen an Querverweisen führt

dann nur zufällig irgendwann einmal tatsächlich wirksamen Code aus. Man könnte jetzt auf die Idee kommen, eine IDE mit integrierter syntaktischer Codesuche zu verwenden, um diesen Quelltexthaufen in Form zu bringen und zu verstehen. Aber das ist vergebens: Auch mit der gesamten Codebasis und einem Index darauf kann man nicht vorhersagen, was eine gegebene Java Codebase tun wird, wenn man sie startet. Es braucht auch noch Konfigurationsdateien. Diese sind ein weiterer Haufen, Properties-Dateien, in einem vorsintflutlichen Vorläufer von YAML geschrieben: XML. Oder jedenfalls ist das, was wir denken sollen: Mit den Properties und der Codebasis können wir endlich versuchen zu verstehen, was Java tut. Und das wäre auch beinahe so, aber JNDI ist genau angetreten, dieses Problem zu beheben: Directory Lookups! Statt also die Anwendung und ihre Konfiguration zu paketieren und dann die Pakete in Produktion zu installieren, können wir nun mit JNDI die Konfiguration vom Netz lesen. Das heißt, die eigentlichen Konfigurationsdateien, die uns sagen, was die Anwendung tut, sind… nicht mehr da. Fortschritt! Das stellt sicher, dass uns niemand mehr hacken kann, weil niemand den Code mehr versteht: Wichtige, zum Verständnis der Codebasis notwendige Informationen sind versteckt in einem Verzeichnisdienst, und wir wissen nicht mal welcher. Aber wir können das noch einen Schritt weiterspielen: Der Code, der den Directory Lookup vornimmt, ist auch nicht da, nur ein Bootstrap: Event and Service Provider Packages [2]. <figure class=„a-inline-image a-u-inline“> </figure> Dank JNDI SPI können wir also Java Classfiles von einem LDAP-Server ausliefern lassen, die angeblich ein Printer-Object generieren, wenn wir nach einem Printer fragen – dann aber stattdessen Doom installieren. Oder einen Kryptominer oder Verschlüsselungstrojaner. So geht Enterprise Security. Der Java-Slogan ist ja „Write Once, Run Everywhere“. Wir haben da viele Witze drüber gemacht, weil Java so oft gecrashed ist – die meisten Java-Anwendungen liefern inzwischen ja die gesamte Ausführungsumgebung einschließlich der JRE mit, damit überhaupt etwas funktioniert. Jetzt funktioniert endlich mal alles – und die Leute sind wieder unglücklich. <h3 class=„subheading“ id=„nav_notabug_wontfix_1“>NOTABUG, WONTFIX</h3> Aber im Ernst: Viele behandeln den log4j-Exploit wie einen Bug, einen Programmierfehler, eine Verletzung einer Spezifikation. Genau das ist jedoch nicht der Fall: Es funktioniert – wortwörtlich – endlich einmal alles wie spezifiziert und dokumentiert: All die Modularität und dynamische Erweiterbarkeit von Java hat ganz wunderbar und genau wie geplant zusammengearbeitet und funktioniert. Darauf haben wir dekadenlang hingearbeitet! „NOTABUG, WONTFIX“ Und das ist das eigentliche Problem hier. Viele rufen jetzt nach „Mehr Kontrolle!“, „Mehr Review!“, „Mehr Funding!“, „Mehr Augen auf den Code!“. Was wirklich helfen würde wäre weniger Code, weniger Indirektion und Boilerplate, und einfach mehr… Einfachheit. Wieso brauche ich ein LogAppenderFactorySingleton, das XML liest, um den Namen der Klasse zu bekommen, die es instanziieren muss, damit ich meine Logzeile da einwerfen kann, um sie asynchron an einen LogStream anzuhängen? Das ist nicht einfach. Was ist einfach? JSON nach stderr drucken. Das ist einfach. Aber Firmen stellen seit etwa einer Dekade Leute ein, die nicht wissen, was stdout und stderr sind und das ist irgendwie okay, weil inzwischen ja sowieso alles ein Webservice ist [3]. Softwareentwicklung ist viel moderner geworden: Wir haben Merge Requests mit Code Review, CI/CD-Pipelines, Infrastructure as Code und Immutable Infrastructure. Das nützt nur nix, wenn meine Java Logging Library auf dem Mars Code von Directory-Servern auf der Erde nachlädt [4] und so das „Remote“ in RCE komplett neu definiert. Die Analyse von Dependencies hat nur dann Sinn, wenn die Liste dieser Abhängigkeiten endlich und genau so immutable wie

die Infrastruktur ist. Java hatte einmal die notwendigen Kontrollknöpfe, mit denen wir erzwingen konnten, dass diese Liste von Abhängigkeiten endlich und unveränderlich ist. Wenn man auf diese Art von Lebensstil steht, gibt es etwas, das sich SM nennt: Oracle SM [5]. SM bringt die notwendigen Verträge und die Disziplin, die eine Codebase braucht. Aber die meisten Anwender stehen nicht auf SM, und lehnen die Idee ab. Also wird die Funktionalität in Java 17 als deprecated (veraltet) gekennzeichnet und später entfernt werden (JEP 411: Deprecate the Security Manager for Removal [6]). <figure class=„a-inline-image a-u-inline“> <figcaption class=„a-caption“>(Bild: imgflip.com)</figcaption> </figure><h3 class=„subheading“ id=„nav_wie_ein2“>Wie ein Dreijähriger</h3> Auch JNDI hat SM abgelehnt und hat ein promiskuitives Interface, das Code irgendwoher lädt und ausführt. Man muss sich das wie einen Dreijährigen vorstellen, der sich jede Klasse in den Mund steckt, um herauszufinden, wie sie schmeckt und ob sie sich ausführen lässt. Das ist am Ende genau die Spezifikation: Hier ist ein Object, deserialisiere es. In Java bedeutet das, dass der Code für die Klasse des Objektes irgendwo her kommen muss, deren Methoden dann ausgeführt werden. Natürlich würde niemand das für eine gute Idee halten, wenn man es so formuliert. Aber andererseits hat es die Person, die das vor acht Jahren implementiert hat, nicht gesehen, und auch die Hunderttausend, die log4j in ihre Codebasis importiert haben, sind nicht darüber gestolpert. Was sagt uns das über den Dependency-Management-Prozess von Organisationen, die Software entwickeln? Über das Verständnis der Codebase, der Abhängigkeiten, und der Prozesse, die Datenfluss und Releases planen? Ja, genau. „Wir verwenden eine modulare Architektur und agile Methoden, um die Entwicklungsgeschwindigkeit zu steigern.“ E = 1/2m*(v^2) Mehr Entwicklungsgeschwindigkeit macht größere Krater. <h3 class=„subheading“ id=„nav_code_ist3“>Code. Ist. Nicht. Dein. Freund.</h3> Ganz besonders nicht dynamisch aus dem Internet nachgeladener Code. Code. Ist. Nicht. Dein. Freund. Ich weiß, es klingt komisch, wenn man Entwickler ist und den eigenen Lebensunterhalt damit bestreitet, dass man glaubt, man sei mit dem Code befreundet. Aber so ist es. Weniger Code ist besserer Code. Am besten so wenig Code, dass man ihn zur Gänze und mit allen seinen Interaktionen verstehen kann. Und auch den Code, der notwendig ist, um den eigenen Code zu betreiben. <header class=„a-boxheader“ data-collapse-trigger=„“>Lesen Sie auch</header><div class=„a-boxtarget a-boxcontent“ data-collapse-target=„“><article class=„a-article-teaser a-article-teaser–horizontal-layout article-teaser–articlebox a-u-no-margin-bottom a-theme“ data-cid=„0“><figure class=„a-article-teaserimage-container“> </figure> <header><h1 class=„a-article-teasertitle a-u-mb-1“>heise-Security-Webinar: Log4j – der Praxis-Ratgeber für Admins</h1></header> [7]</article> () <hr /> URL dieses Artikels:
<small><code>https://www.heise.de/-6294476</code></small> Links in

diesem Artikel:
<small><code>[1] https://docs.oracle.com/javase/tutorial/jndi/overview/index.html</code></small>
<small><code>[2] https://docs.oracle.com/javase/tutorial/jndi/overview/event.html</code></small>
<small><code>[3] https://twitter.com/apenwarr/status/1469183890749558784</code></small>
<small><code>[4] https://twitter.com/TheASF/status/1400875147163279374</code></small>
<small><code>[5] https://docs.oracle.com/javase/tutorial/essential/environment/security.html</code></small>
<small><code>[6] https://openjdk.java.net/jeps/411</code></small>
<small><code>[7] https://www.heise.de/news/heise-Security-Webinar-Log4j-der-Praxis-Ratgeber-fuer-Admins-6294236.html</code></small>
<small><code>[8] mailto:map@ix.de</code></small>
 Copyright © 2021 Heise Medien </html>