Detecting Code Injection at Build Time

u5551869

Contents

1 Background and Research

One of the biggest attacks in recent years was the Solarwinds attack. It was a supply chain attack, meaning the affected parties where compromised using a backdoor present in a third party software (the Orion system). Supply chain attacks often have significant and far reaching impacts as the vulnerable software is often used by several companies or integrated into other programs [2].

1.1 Impact of the Solarwinds Attack

The attack carried out by the Russian cyber group Nobelium, had severe and long lasting impacts, with the backdoor going undetected for over a year and over 18,000 customers installing the affected software, including 9 government agencies [3]. On average the Solarwinds attack cost affected companies 11% of their annual turnover and caused the Solarwinds stock price to fall 23% in a week [1].

1.2 Sunspot Malware

To most effective method to mitigate supply chain attacks is for the threats to be dealt with at the source before it can spread up the supply chain, as such this section will focus on how the backdoor was introduced into the Orion system. The malware responsible for planting the backdoor into the Orion system was sunspot. Sunspot inserted the backdoor by hijacking the build process for the Orion software[4].

To hijack the build process Sunspot first gives itself the SeDebugPrivilege this allows the malware to access other process's memory and their process information. Sunspot then starts to monitor for the MSBuild process, the process used by visual studio to compile projects. It does this by generating a hash of each running process using a custom hash function and comparing it to the hash of "MSBuild.exe", this is done to obscure what the Sunspot process is doing in order to avoid detection by end point security solutions and anti viruses. After an MSBuild process has been found, Sunspot gets the process's PEB (Process Environment Block) by using the NtQueryInformationProcess win32 api call. The PEB contains a pointer to the RTL_USER_PROCESS_PARAMETERS structure which contains the command line string passed to the process, this is used by the Sunspot malware to locate the file path of the source code being compiled[5].

After the file path to the source code has been retrieved, Sunspot replaces the legitimate source code with the compromised code. This involves four steps, first InventoryManager.cs is backed up to InventoryManager.bk, next Sunspot decrypts the backdoored code it has stored and writes it to InventoryManager.tmp, then InventoryManager.tmp is written to InventoryManager.cs, finally after the solution has been built the backdoored files are removed and InventoryManager.bk is moved back to InventoryManager.cs. This leaves no trace that the source code was altered before it was compiled [4].

1.3 Need for Build Time Injection Detection

The Solarwinds attack and the Sunspot malware used in it show a clear lack of security around code injection during software building. As mentioned in the previous section Sunspot leaves no evidence behind after the build has finished this makes checking integrity hashes of files before and after compilation pointless because they will still be the same, as the changes to the files are reversed right after the build has finished. As such it is clear that a tool is needed for detecting alterations to files during compilation.

1.4 Justification for Programming Language and Paradigms Used

This tool lends its self well to an object oriented approach, as tool will involve the monitoring of files, directories and processes, all of which can be represented as an object. Information about the processes, directories and files can then be held in the attributes of the object that represents them.

The tool also has a highly event driven nature and as such lends itself to the event driven programming paradigm. The event driven paradigm is orthogonal meaning it can be used in conjunction with other paradigms such as OOP. The tool suits the event driven paradigm well as it must monitor and respond

to events such as build processes starting, or directories being written or altered. As the tool is suited to both event driven programming and object oriented programming I have decided to use Java as it is object oriented and has many established design patterns for event driven programming.

Java provides the 'Java native access' library, a well maintained library for interfacing with the win32 API which is necessary when interacting with windows resources such as files, directories and processes. JNA also provides classes to abstract the data structures (contiguous byte arrays) returned by and required by the win32 API into Java objects this makes interacting with the win32 API much easier.

2 Tool Implementation and Justification

2.1 High Level Overview

At a high level the tool has to carry out three functions, first is to detect 'msbuild', the second is to detect changes to source files, and the third is to alert the user to changes to source files during compilations. This would addresses the issue of code injection at build time identified in the previous section. Additional features such as SIEM integration would greatly enhance the utility of the tool.

2.2 Justification for Event Driven Programming and Pub Sub Architecture

As mentioned in section 1.3 the tool lends itself well to an event driven approach, event driven programming consists of three components event producers, events and event consumers. Event producers are responsible for generating events, in the case of this tool the 'DirectoryWatcher' and 'ProcessWatcher' classes are the event producers. Events represent significant changes in state, events this tool monitors for include changes to the content of directories and the starting and stopping of build processes. Event consumers listen for events and often execute an event handler in response to the event. An event consumer within my tool is the userAlert class which is responsible for alerting the user when a alert event occurs during a build process.

A pub sub architecture has been used to handle communication between event consumers and event producers. To facilitate objects subscribing to, and publishing events, an event bus pattern has been used, this helps to decouple event sources and event handlers. This decoupling of event sources and handlers allows for smoother handling of asynchronous events which is necessary as the tool monitors for both build processes and directory changes simultaneously as such it needs to be able to handle asynchronous events. Decoupling also has the benefit of modularity, handlers/consumers can be altered, removed or added without impacting the event sources and other handlers, this makes maintenance and future development easier [6].

2.3 Security Implications of Using Java

One key feature of Java is its garbage collector this is responsible for freeing memory. The garbage collector enhances the security of Java as it reduces the chance of buffer overflow, use after free and other memory based attacks. Another feature of Java that increases security is Java's strong typing, Java enforces type checking at compile time and runtime which mitigates type confusion attacks. Java also forces handling of checked exceptions, if a checked exception is not handled then an error will be thrown at compile time this helps to limit vulnerabilities introduced due to the lack of sufficient error handling [7].

2.4 Implementation of Core Functionality

2.4.1 Events

Events in Java event driven programming are represented using objects of an event class. My tool has a base event class that is has multiple sub classes for each event that the tool produces each sub class adds different attributes to the base class as each event type has different relevant information for example the filename is relevant information for a file event but not a build event. The base class also provides the 'toMessageMap' function which converts the attributes of the event object into a key value map so that it can be parsed as

a message into a logger. The final modifier is used to protect the attributes of the event object from being altered after the event has been created.

```java
public static class Event {  7 usages  3 inheritors
    final Date timestamp;  1 usage
    final String eventType;  3 usages
    public Event(String eventType){  3 usages
        this.timestamp = new Date();
        this.eventType = eventType;
    }
    public MapMessage toMapMessage() {  2 usages
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            return new MapMessage(objectMapper.convertValue(this, new TypeReference<Map<String, Object>>() {}));
        } catch (Exception e) {
            return new MapMessage(new HashMap<String, Object>());
        }
    }
}
```

Figure 1: Event Base Class with toMapMessage Function

```java
public static class BuildEvent extends Event {  5 usages
    final String action;  3 usages
    final String buildProcess;  2 usages
    final int processID;  1 usage
    public BuildEvent( String action, String buildProcess, int processID){
        super( eventType: "BuildEvent");
        this.action = action;
        this.buildProcess = buildProcess;
        this.processID = processID;
    }
}
```

Figure 2: Build event inherits from event, used to represent a build process stopping/starting

```java
public static class FileEvent extends Event {  3 usages
    final String fileName;  4 usages
    final String action;  1 usage
    public FileEvent(String fileName, String action){
        super( eventType: "FileEvent");
        this.fileName = fileName;
        this.action = action;
    }
}
```

Figure 3: File Event inherits from event, used to represent file alterations

```
public static class AlertEvent extends Event {  4 usages
    final String alteredFile;  2 usages
    final ArrayList<String> runningPrivilegedProcesses;  1 usage
    public AlertEvent(String fileName, ArrayList<String> privilegedProcesses) {
        super( eventType: "AlertEvent");
        this.alteredFile = fileName;
        this.runningPrivilegedProcesses = privilegedProcesses;
    }
}
```

Figure 4: Alert event inherits from event, used to signify the user should be alerted

2.4.2 Event Bus

This class follows an event bus pattern and allows other objects to subscribe to events and publish events. Each event type is stored as a key in a hash map with the value assigned to each key being a list of listeners that have subscribed to that event. When an event is published the event bus checks the event type of the event and retrieves the list of listeners that have subscribed to that event type, it then iterates over the list calling the event handler (onEvent) for each event listener. The subscribe function gets the listener list for the provided event type from the listeners hash map. If the provided event type isn't in the listeners hash map then it is added with an empty list as its value. The provided listener object is then added to the list.

```
public class EventBus {  12 usages
    private final Map<String, List<EventListener>> listeners = new HashMap<>();  4 usages

    public void subscribe(String eventType, EventListener listener) {  4 usages
        listeners.computeIfAbsent(eventType,  String k -> new ArrayList<>()).add(listener);
    }
     public synchronized void publish(Events.Event e) {  4 usages
        new Thread(() -> {
            List<EventListener> eventListeners = listeners.getOrDefault(e.eventType, List.of());
            for (EventListener listener : eventListeners) {
                listener.onEvent(e);
            }
        }).start();

    }
    public void unsubscribe(String eventType, EventListener listener) {  no usages
        List<EventListener> eventListeners = listeners.getOrDefault(eventType, List.of());
        eventListeners.remove(listener);
    }
    public List<String> getAllEventTypes() {  1 usage
        return new ArrayList<>(listeners.keySet());
    }
}
```

Figure 5: Event Bus class

2.4.3 Event Listener Interface

This interface is used to create classes that need to listen for events. It ensures that all listeners provide an event handler in the form of an onEvent function. Generic types are a feature of Java and have been used so that the onEvent function of listeners can specify which subclass of event they accept as an argument.

```
public interface EventListener<T extends Events.Event> {  10 usages  4 implementations
    void onEvent(T e);//on event takes any object that is a subclass of the event class  1 usage  4 implementations
}
```

Figure 6: Interface for Listener classes

2.4.4 Monitoring for Build Processes

In order to detect injection of code during build processes the tool first has to be able to detect build processes starting and stopping. Processes are managed by the OS and therefore handles to running processes can only be obtained through the OS, in order to get a list of all running processes and their names I have used the JNA library to interface with the win32 API and other windows dlls.

The process watcher class schedules a function to run ever 0.01 seconds. This function retrieves all running build processes then compares the running build processes with the build processes that where running last time the function was called, for each new build processes a build event is published with the action attribute set to 'BuildStart' for each build process missing a build event with a 'BuildStopped' action attribute is sent.

```
public class ProcessWatcher {  1 usage
    private final EventBus bus;  3 usages
    private Set<String> buildProcesses;  2 usages
    private Set<String> runningBuildProcesses;  4 usages

    public ProcessWatcher(EventBus bus, Set<String> buildProcesses) {
        this.bus = bus;
        this.buildProcesses = buildProcesses;
        this.runningBuildProcesses = new HashSet<String>();
    }
```

Figure 7: Declaration and constructor for ProcessWatcher class

```
public void start() {  1 usage
    Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
        var running = getRunningBuildProcesses();
        for (var proc: running.toArray()) {
            if( !this.runningBuildProcesses.contains(proc.toString())){
                this.bus.publish(new Events.BuildEvent( action: "BuildStart", proc.toString(),Integer.valueOf(proc.toString().split( regex: "/")[1])));
            }
        }
        for (var proc: this.runningBuildProcesses.toArray()) {
            if( !running.contains(proc.toString())){
                this.bus.publish(new Events.BuildEvent( action: "BuildStopped", proc.toString(),Integer.valueOf(proc.toString().split( regex: "/")[1])));
            }
        }
        this.runningBuildProcesses = running;
    }, initialDelay: 0, period: 10, TimeUnit.MILLISECONDS);
}
```

Figure 8: Start function of the ProcessWatcher class schedules process monitoring

The getRunningBuildProcesses function uses the CreateToolhelp32Snapshot function from the kernel32 dll to get a handle to a snapshot containing the process ID and filenames of all running processes. It then iterates over each process in the snapshot and adds all the build processes to a set and returns the set.

```
private Set<String> getRunningBuildProcesses() {  1 usage
    Set<String> runningBuildProcessList = new HashSet<>();
    WinNT.HANDLE snapshot = Kernel32.INSTANCE.CreateToolhelp32Snapshot(
            new WinDef.DWORD( value: 2), new WinDef.DWORD( value: 0));
    if (WinBase.INVALID_HANDLE_VALUE.equals(snapshot)) {
        System.err.println("Failed to take process snapshot");
        return new HashSet<>();
    }
    Tlhelp32.PROCESSENTRY32 entry = new Tlhelp32.PROCESSENTRY32();
    if (!Kernel32.INSTANCE.Process32First(snapshot, entry)) {
        System.err.println("Failed to get first process");
        Kernel32.INSTANCE.CloseHandle(snapshot);
        return new HashSet<>();
    }
    do {
        if (this.buildProcesses.contains(Native.toString(entry.szExeFile).toLowerCase())){
            runningBuildProcessList.add(Native.toString(entry.szExeFile).toLowerCase()+"/"+entry.th32ProcessID.toString());
        }
    } while (Kernel32.INSTANCE.Process32Next(snapshot, entry));
    Kernel32.INSTANCE.CloseHandle(snapshot);
    return runningBuildProcessList;
}
```

Figure 9: The getBuildProcesses function of the ProcessWatcher class

2.4.5 Monitoring Directory Changes

The build listener class consumes build events and keeps track of the number of running build processes. For each build event with the action attribute set to BuildStart the startWatching function is called, if no other build processes are running then a new thread is created a directory watcher object is instantiated and its watch function called, if other build processes are already running the number of running build processes are incremented and the function returns. If the action attribute is set to BuildStopped then stopWatching is called this decrements the number of running build processes and sets the watch flag to false if no build processes are running, this signals to the thread that is running the directory watcher to stop watching and finish. Although watch flag is a resource shared by two threads the flag attribute is an atomic boolean and is only accessed using the get() method and only set with getAndSet() method these are atomic actions and ensure that only one watchThread is running at a time.

```
public class BuildListener implements EventListener<Events.BuildEvent> {  1 usage
    private final ArrayList<String> dirs;  2 usages
    private final EventBus bus;  3 usages
    private Thread watchThread;  2 usages

    public BuildListener(ArrayList<String> dirs, EventBus bus) {  1 usage
        this.dirs = dirs;
        this.bus = bus;
        this.bus.subscribe( eventType: "BuildEvent",  listener: this);
    }
```

Figure 10: Declaration and constructor for BuildListener class

```java
@Override  1 usage
public void onEvent(Events.BuildEvent e) {
    if (e.action.equals("BuildStart")) startWatching(e);
    else if (e.action.equals("BuildStopped")) stopWatching();
}
```

Figure 11: Event handler for build events in the BuildListener class

```java
private void startWatching(Events.BuildEvent buildProcess) {  1 usage
    WatchFlag watching = WatchFlag.getInstace();
    ++watching.processesWatched;
    if(watching.flag.getAndSet(true)) {
        return;
    }
    Thread watchThread = new Thread(() -> {
        try {
            DirectoryWatcher dw = new DirectoryWatcher(this.dirs, bus, watching);
            dw.watch(); // Watch while flag is true
        } catch (Exception e) {
            System.out.println("Watcher error: " + e.getMessage());
        }
    });
    watchThread.start();
    System.out.println("Build Process Started: " + buildProcess.buildProcess);
}
```

Figure 12: The startWatching function of the BuildListener class

The DirectoryWatcher class is an event producer that publishes a file event when it detects an alteration made to files within the specified directories and the watch flag is set to true. It monitors changes to files in directories by obtaining the handles for the directories using the CreateFileW function in the kernel32 dll, and then uses the ReadDirectoryChangesW function from the same dll to get information on any file writes or name alterations that have occurred. If an alteration occurs to a file the directory watcher publishes a file event with the name of the altered file being stored in the fileName attribute of the FileEvent object.

```java
public class DirectoryWatcher {  2 usages
    private final ArrayList<String> directories;  3 usages
    private final EventBus bus;  2 usages
    private final WatchFlag watching;  3 usages
    public DirectoryWatcher(ArrayList<String> directories, EventBus bus, WatchFlag watching) {
        this.directories = directories;
        this.bus = bus;
        this.watching = watching;
    }
```

Figure 13: Declaration and constructor for the DirectoryWatcher class

```java
public void watch() {  1 usage
    ArrayList<WinAPI.HANDLE> handles = new ArrayList<>();
    for (String directory : directories) {
        WinAPI.HANDLE hDir = WinAPI.INSTANCE.CreateFileW(
                new WString(directory),
                WinAPI.FILE_LIST_DIRECTORY,
                dwShareMode: WinAPI.FILE_SHARE_READ | WinAPI.FILE_SHARE_WRITE | WinAPI.FILE_SHARE_DELETE,
                lpSecurityAttributes: null,
                WinAPI.OPEN_EXISTING,
                WinAPI.FILE_FLAG_BACKUP_SEMANTICS,
                hTemplateFile: null
        );
        if (hDir == null || hDir.getPointer() == Pointer.NULL) {
            System.out.println("Failed to open directory handle.");
        }
        else{
            handles.add(hDir);
        }
    }
```

Figure 14: The watch function of the DirectoryWatcher class pt.1

```java
    Pointer bufferPtr = new Memory( size: 1024);
    System.out.println("Monitoring for file write events in: " + this.directories);

    while (this.watching.flag.get()) {
        IntByReference bytesReturned = new IntByReference();

        for (WinAPI.HANDLE hDir : handles) {
            boolean success = WinAPI.INSTANCE.ReadDirectoryChangesW(
                    hDir,
                    bufferPtr,
                    nBufferLength: 1024,
                    bWatchSubtree: true,
                    dwNotifyFilter: WinAPI.FILE_NOTIFY_CHANGE_LAST_WRITE | WinAPI.FILE_NOTIFY_CHANGE_FILE_NAME,
                    bytesReturned,
                    lpOverlapped: null,
                    lpCompletionRoutine: null
            );
            if (!this.watching.flag.get()){
                break;
            }
            if (success) {
                byte[] bytes = bufferPtr.getByteArray( offset: 0,bytesReturned.getValue());
                int fileNameLength = bufferPtr.getInt( offset: 8);
                byte[] fileNameBytes = new byte[fileNameLength];
                System.arraycopy(bytes, srcPos: 12,fileNameBytes, destPos: 1, length: fileNameLength-1);
                String fileName = new String(fileNameBytes, StandardCharsets.UTF_16);
                this.bus.publish(new Events.FileEvent(fileName,  action: "FileAltered"));
            } else {
                System.out.println("Error reading directory changes.");
            }
        }
    }
    for (WinAPI.HANDLE hDir : handles) {
        WinAPI.INSTANCE.CloseHandle(hDir);
    }
}
```

Figure 15: The watch function of the DirectoryWatcher class pt.2

2.4.6 Alerting the User

When a file event occurs the injection listener class's onEvent function gets called this checks the file type of the altered file against the file types in the config file, if it is in the config file then an alert event is published.

```
public class InjectionListener implements EventListener<Events.FileEvent> {  1 usage  Ryan Ballard
    private final Set<String> fileExtensions;  3 usages
    EventBus bus;  3 usages

    public InjectionListener(EventBus bus, Set<String> fileExtensions) {  1 usage  Ryan Ballard
        this.bus = bus;
        this.bus.subscribe( eventType: "FileEvent",  listener: this);
        this.fileExtensions = fileExtensions;
    }

    @Override  1 usage  Ryan Ballard
    public void onEvent(Events.FileEvent e) {
        if (this.fileExtensions.contains(e.fileName.split( regex: "[.]")[e.fileName.split( regex: "[.]").length-1].toLowerCase()) || this.fileExtensions.contains("*")) {
            ArrayList<String> privilegedProcesses =  SeDebugPrivilegeChecker.getPrivilegedProcesses();
            Events.AlertEvent alert = new Events.AlertEvent(e.fileName.toLowerCase(), privilegedProcesses);
            this.bus.publish(alert);
        }
    }
}
```

Figure 16: Injection listener class publishes an alert event if file event file type matches a configured file type

When an alert event is published the user alert class's event handler is called. This event handler then creates an alert window to notify the user that a source file has been altered during the build process and which file was altered.

```
public class UserAlert implements EventListener<Events.AlertEvent> {  1 usage

    public UserAlert(EventBus bus) {  1 usage
        bus.subscribe("AlertEvent", this);
    }

    public void createAlert(String fileName) {  1 usage
        JOptionPane.showMessageDialog(null, "File "+fileName+" was altered during compilation", "Alert", JOptionPane.INFORMATION_MESSAGE);

    }
    public void onEvent(Events.AlertEvent e) {  1 usage
        this.createAlert(e.alteredFile);
    }
}
```

Figure 17: User alert class notifies user of alterations to a file during build time

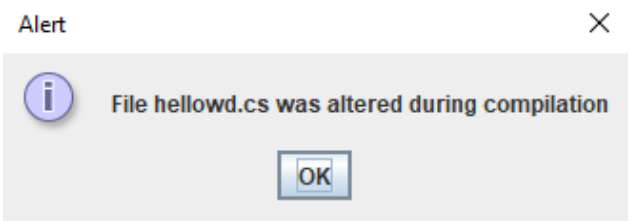

Figure 18: User Alert Pop Up

2.5 Additional Functionality

As mentioned in section 2 the use of pub sub architecture and the decoupling of producers and consumers allows for new features to be developed and added without impacting other consumers or producers. This

means that developing the additional features in this section required almost no alteration to any of the code in the previous section.

2.5.1 Logging Events to Logstash for SIEM Integration

For many companies its important that events such as source code injection during build time is visible in their SIEM. This allows them to monitor patterns in these events and identify affected devices. To achieve integration with SIEMs the tool logs data over a TCP connection to a log aggregator. Compatibility with ELK stack (Elasticsearch, Logstash, Kibana) has been tested, support for other SIEM software has not been confirmed. SIEM integration although not a core part of the tools functionality drastically increases visibility and security monitoring within a multi endpoint environment such as a software development company.

The SIEM integration is handled using log4j, this is a Java logging utility and can be configured to write logs over a TCP port. This is used to send logs to a log aggregator such as Logstash. Logs are also held on device as well if integration with a log aggregator is not necessary. I explicitly checked the version of log4j before using it to make sure it was not vulnerable to log4shell.

```
<Appenders>
 <Socket name="Logstash" host="localhost" port="5000" protocol="TCP">
  <JsonTemplateLayout eventTemplateUri="classpath:LogstashJsonEventLayoutV1.json">
    <EventTemplateAdditionalField key="HostName" value="${hName}"/>
    <EventTemplateAdditionalField key="HostAddress" value="${hIpAddress}"/>
  </JsonTemplateLayout>
 </Socket>
```

Figure 19: Log4j configuration file creates appender for writing to TCP port

```
public class logger implements EventListener<Events.Event> {  1 usage

    public logger(EventBus bus) {  4 usages
        List<String> eventTypes =  bus.getAllEventTypes();
        eventTypes.forEach( String eventType -> {bus.subscribe(eventType, this);});

    }
    public void onEvent(Events.Event e) {  1 usage
        Logger log = LogManager.getLogger("root");
        try {
            if ("BuildEvent".equals(e.eventType)) {
                if (log.isInfoEnabled()){
                    log.info(e.toMapMessage());
                }

            } else {
                if (log.isWarnEnabled()){
                    log.warn(e.toMapMessage());
                }
            }
        }catch (org.apache.logging.log4j.core.appender.AppenderLoggingException err) {
            System.err.println("Could not connect to logstash only logging locally");
        }
    }
}
```

Figure 20: logger class subscribes to all events and logs the event when on event called

```java
public static void main(String[] args) {
    try{
        InetAddress localHost = InetAddress.getLocalHost();
        System.setProperty("hostName", localHost.getHostName());
        System.setProperty("hostAddress", localHost.getHostAddress());

    }catch(Exception e){
        System.setProperty("hostName", "unknown");
        System.setProperty("hostAddress", "0.0.0.0");
    }
    System.setProperty("log4j2.configurationFile", "./log4j2.xml");
    try{
        Logger log = LogManager.getLogger( name: "root");
        log.info("SecureBuildStarting");
    }catch(AppenderLoggingException e){
        System.out.println("Could not connect to logstash only logging locally");
    }
```

Figure 21: Setting logger configurations and host name and address

2.5.2 Detecting Running Processes with the seDebug Privilege

Although not all source code injection attacks are done by privileged processes the Solarwinds attack highlighted in section 1 used Sunspot malware which ran with the seDebug Privilege, this is what allowed it to extract the directory of the source code from the msbuild process. To help identify a possible subset of processes that have the privilege to carry out such an operation the tool also generates a list of all processes on the system running with seDebugPrivileges.

```java
public static ArrayList<String> getPrivilegedProcesses() {  1 usage
    HANDLE snapshot = Kernel32.INSTANCE.CreateToolhelp32Snapshot(
            new WinDef.DWORD( value: 2), new WinDef.DWORD( value: 0));
    if (WinBase.INVALID_HANDLE_VALUE.equals(snapshot)) {
        System.err.println("Failed to take process snapshot");
        return new ArrayList<>();
    }
    Tlhelp32.PROCESSENTRY32 entry = new Tlhelp32.PROCESSENTRY32();
    if (!Kernel32.INSTANCE.Process32First(snapshot, entry)) {
        System.err.println("Failed to get first process");
        Kernel32.INSTANCE.CloseHandle(snapshot);
        return new ArrayList<>();
    }
    ArrayList<String> privilegedProcesses = new ArrayList<>();
    do {
        HANDLE hProcess = Kernel32.INSTANCE.OpenProcess(
                WinNT.PROCESS_QUERY_INFORMATION,
                b: false,
                entry.th32ProcessID.intValue()
        );
        if (hProcess != null) {
            boolean hasDebug = processHasSeDebugPrivilege(hProcess);
            if (hasDebug) {
                privilegedProcesses.add("Process [" + entry.th32ProcessID.intValue() + "] " + Native.toString(entry.szExeFile));
            }
            Kernel32.INSTANCE.CloseHandle(hProcess);
        }
    } while (Kernel32.INSTANCE.Process32Next(snapshot, entry));
    Kernel32.INSTANCE.CloseHandle(snapshot);
    return privilegedProcesses;
}
```

Figure 22: The getPrivilegedProcesses function of the seDebugChecker class enumerates processes

To identify privileged processes the tool enumerates all processes on the system opens a handle to their access token then retrieves the privileges of the process using the handle to the access token. Accessing other processes access tokens requires high level privileges (seDebugPrivilege) this means not only does the tool need to be run as administrator it needs to also assign itself the seDebug privilege as not all processes run by administrators have the seDebug privilege by default. To do this the process enabler class has been created this class provides the static enableSeDebugPrivilege function that retrieves the access token for the current process its running in then modifies the privileges in the access token to enable the seDebug privilege.

```
public static boolean processHasSeDebugPrivilege(HANDLE hProcess) {  1 usage
    HANDLEByReference hToken = new HANDLEByReference();
    if (!Advapi32.INSTANCE.OpenProcessToken(hProcess, WinNT.TOKEN_QUERY, hToken)) {
        return false;
    }
    IntByReference size = new IntByReference();
    Advapi32.INSTANCE.GetTokenInformation(
            hToken.getValue(),
            WinNT.TOKEN_INFORMATION_CLASS.TokenPrivileges,
             structure: null,
             i1: 0,
            size
    );
    WinNT.TOKEN_PRIVILEGES privileges = new WinNT.TOKEN_PRIVILEGES( nbOfPrivileges: (size.getValue()-4)/12);
    boolean result = Advapi32.INSTANCE.GetTokenInformation(
            hToken.getValue(),
            WinNT.TOKEN_INFORMATION_CLASS.TokenPrivileges,
            privileges,
            size.getValue(),
            size
    );
    if (!result) {
        Kernel32.INSTANCE.CloseHandle(hToken.getValue());
        return false;
    }
    int privilegeCount = (size.getValue()-4)/12;
    WinNT.LUID luid = new WinNT.LUID();
    if (!Advapi32.INSTANCE.LookupPrivilegeValue( s: null, WinNT.SE_DEBUG_NAME, luid)) {
        Kernel32.INSTANCE.CloseHandle(hToken.getValue());
        return false;
    }
    for (int i = 0; i < privilegeCount; i++) {
        WinNT.LUID_AND_ATTRIBUTES entry = privileges.Privileges[i];
        entry.read();
        if (entry.Luid.HighPart == luid.HighPart && entry.Luid.LowPart == luid.LowPart && entry.Attributes.intValue() == 2) {
            Kernel32.INSTANCE.CloseHandle(hToken.getValue());
            return true;
        }
    }
    Kernel32.INSTANCE.CloseHandle(hToken.getValue());
    return false;
```

Figure 23: The hasSeDebugPrivilege function of the seDebugChecker class

```java
public class PrivilegeEnabler {  1 usage
    public static void enableSeDebugPrivilege() {  1 usage
        WinNT.HANDLEByReference hToken = new WinNT.HANDLEByReference();
        // Open the current process token
        if (!Advapi32.INSTANCE.OpenProcessToken(
                Kernel32.INSTANCE.GetCurrentProcess(),
                WinNT.TOKEN_ADJUST_PRIVILEGES | WinNT.TOKEN_QUERY,
                hToken)) {
            System.err.println("Failed to open process token.");

        }
        // Lookup LUID for SeDebugPrivilege
        WinNT.LUID luid = new WinNT.LUID();
        if (!Advapi32.INSTANCE.LookupPrivilegeValue(null, WinNT.SE_DEBUG_NAME, luid)) {
            System.err.println("Failed to lookup privilege value.");
        }
        // Enable the seDebug
        WinNT.TOKEN_PRIVILEGES tp = new WinNT.TOKEN_PRIVILEGES(1);
        tp.Privileges[0] = new WinNT.LUID_AND_ATTRIBUTES(luid, new WinDef.DWORD(WinNT.SE_PRIVILEGE_ENABLED));
        tp.write();
        if (!Advapi32.INSTANCE.AdjustTokenPrivileges(
                hToken.getValue(),
                false,
                tp,
                0,
                null,
                null)) {
            System.err.println("Failed to adjust token privileges.");
        }
        int error = Kernel32.INSTANCE.GetLastError();
        if (error == WinError.ERROR_NOT_ALL_ASSIGNED) {
            System.err.println("SeDebugPrivilege was not assigned. You may need to run as administrator.");
        }
        else{
            System.out.println("SeDebugPrivilege successfully enabled.");
        }
    }
}
```

Figure 24: Privilege enabler class gives Java application seDebug Privilege

3 Security Audit and Testing

3.1 Dependency Vulnerability Analysis

One common ways that vulnerabilities end up in software is through the use of vulnerable dependencies. To mitigate this dependencies should be updated and dependency vulnerability scans conducted periodically. Below shows the output of the dependency vulnerability scan using InteliJ this shows a DOS vulnerability in the version of Jackson I have used. To address this I have updated the Jackson library to the latest version. After updating the Jackson dependencies, dependency analysis still shows a vulnerable dependency within maven-shade this can safely be ignored as maven-shade is a plug-in used at build time not runtime so it wont be included in the tool's jar file.

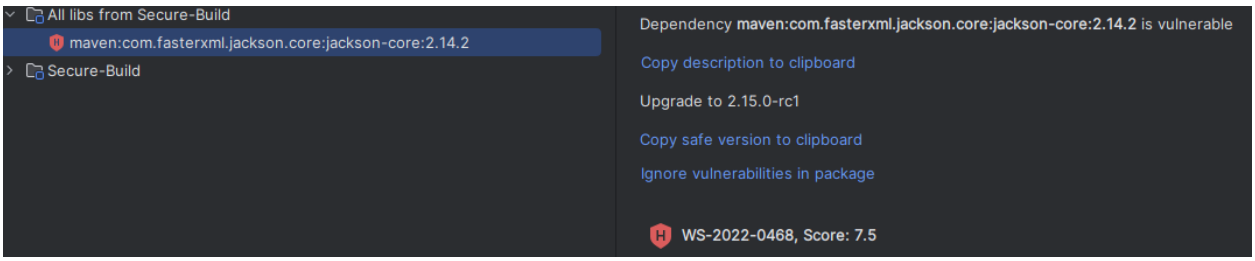

Figure 25: Dependency Analysis shows Vulnerable Jackson Dependency

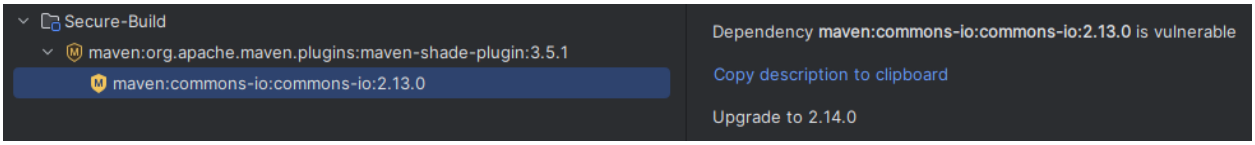

Figure 26: Dependency Analysis After Updating Jackson Dependency

3.2 Static Code Analysis

Static code analysis analyses source files for vulnerabilities. I have used two static analysis tools PMD which is focused on locating unreachable code, unused variables and libraries, and naming convention violations, and infer. Infer is security oriented and is used to locate null pointer exceptions, resource leaks, reachability, missing lock guards, and concurrency race conditions.

Static analysis with infer shoes no issues where detected whilst analysis with PMD shows multiple. Most issues highlighted by PMD static analysis have been addressed, however some have not. First is tight coupling as mentioned earlier the pub sub architecture allows consumers and producers to be developed independently so although tight coupling is not Java best practice it will have little impact on future development and no impact on security. Second the use of hard coded IP addresses this is due to the hardcoded IP address 0.0.0.0 that is only used as a fallback if the application cannot get the IP address of the computer. The close resource error is irrelevant as the Memory object has no close method and is collected by the garbage collector after the function has returned. Fourth unnecessary imports, this is just incorrect the import is used extensively. Finally the naming convention violation can't be helped as the functions identified are named after the underlying win API functions they are bound to.

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  36.363 s
[INFO] Finished at: 2025-05-11T20:50:55+01:00
[INFO] ------------------------------------------------------------------------
Found 15 source files to analyze in /mnt/c/Users/RyanM/IdeaProjects/Secure-Build/infer-out
109/109 [########################################################################] 100% 4.82s

  No issues found
```

Figure 27: Output for infer static analysis shows no issues

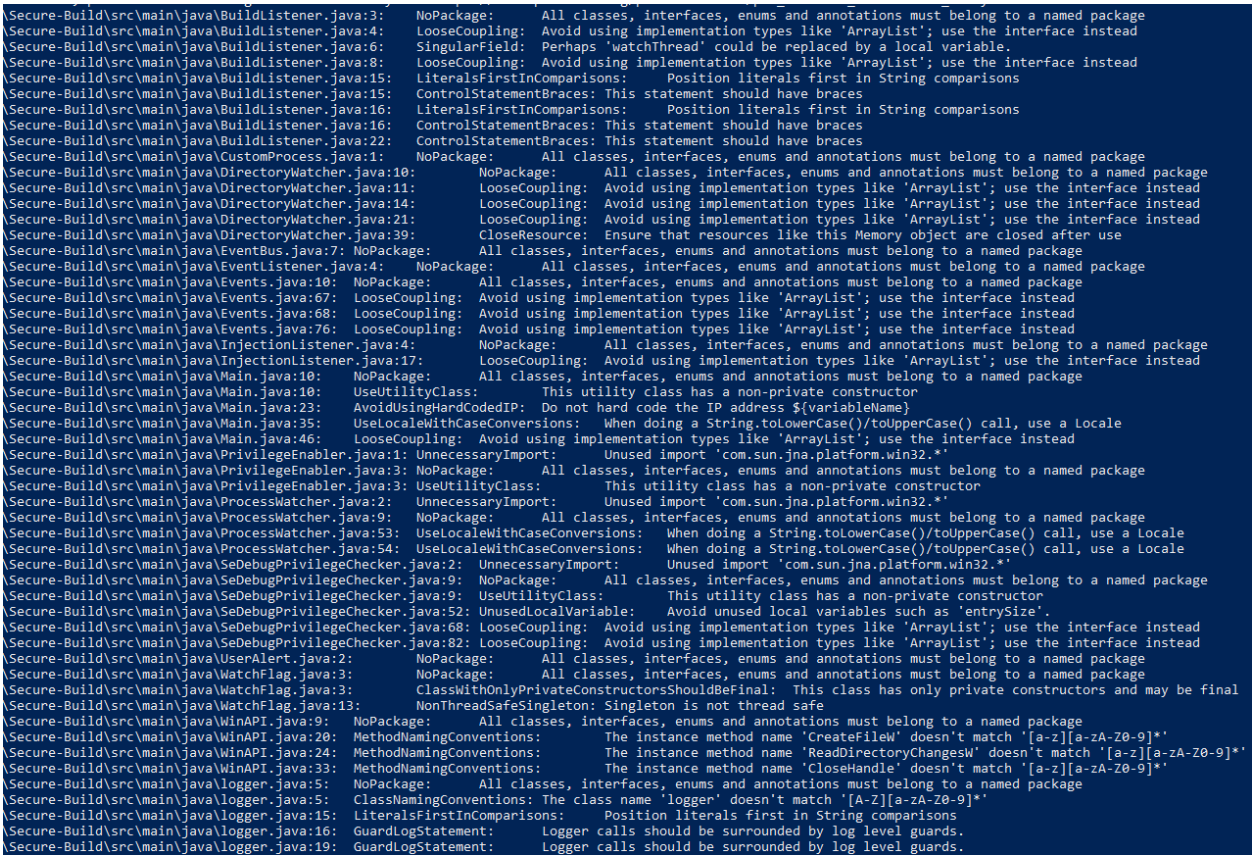

Figure 28: PMD static analysis output

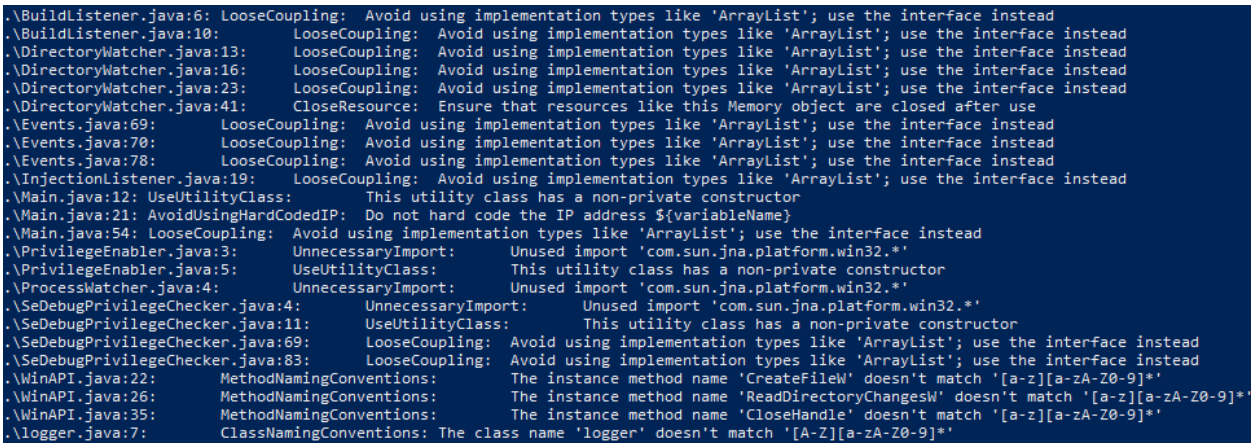

Figure 29: PMD static analysis after fixes

3.3 Black Box Functionality Testing

This section will black box testing of functionality to ensure that a range of possible inputs to the program give expected outputs. The program does not take direct input from the user as it is designed to run in the background, instead it loads tool configurations from the config file and logging configurations from log4j2.xml. Therefore the black box testing will be conducted by altering the config files and observing the tools behaviour.

3.3.1 Testing Build Process Monitoring

No.	Test	Description	Pass result	Pass y/n
1	Detecting build processes	Configure tool to watch for msbuild processes then run an msbuild process and see if it picks it up	Should detect the build process starting	y
2	Detecting large amounts of running build processes running simultaneously	Add a browser as a build process in the config file and open a browser with many tabs to simulate many build processes starting simultaneously	Should detect each build process starting	y
3	Detect short lived build process	Run a msbuild on a simple file that takes less than a second to build	Start and end of build process detected	y
4	Detecting the end of a build process	Execute a build process and check that the tool logs a build start and a build stopped event	Logs should contain a build start and a build stopped event	y
5	Detecting different build processes	Add 2 different build processes to the config file and test if the tool can detect each of them stopping and starting	Logs should contain build stop and start for the different build processes that where configured and executed	y

```
cd D:\Documents\Test\Source
START msbuild .
START make .
```

Figure 30: Power shell script for test No.1,3,4,5

```
build_processes = MSBuild.exe, make.exe
dirs = D:/Documents/Test/Source
fileTypes = cs, cpp
```

Figure 31: Secure Build Config File for test No.1,3,4,5


```
{"@version":1,"source_host":"MSI","message":"action=\"BuildStart\" buildProcess=\"msbuild.exe/19804\" eventType=\"BuildEvent\"","thread_name":"Thread-0","@timestamp":"2025-05-12T13:10:33.991+0100","level":"INFO","logger_name":"root"}
{"@version":1,"source_host":"MSI","message":"action=\"BuildStart\" buildProcess=\"make.exe/13908\" eventType=\"BuildEvent\"","thread_name":"Thread-2","@timestamp":"2025-05-12T13:10:33.991+0100","level":"INFO","logger_name":"root"}
{"@version":1,"source_host":"MSI","message":"action=\"BuildStopped\" buildProcess=\"make.exe/13908\" eventType=\"BuildEvent\"","thread_name":"Thread-3","@timestamp":"2025-05-12T13:10:34.075+0100","level":"INFO","logger_name":"root"}
{"@version":1,"source_host":"MSI","message":"eventType=\"FileEvent\" fileName=\"obj\\Debug\\TestProj.csproj.FileListAbsolute.txt\"","thread_name":"Thread-4","@timestamp":"2025-05-12T13:10:35.155+0100","level":"WARN","logger_name":"root"}
{"@version":1,"source_host":"MSI","message":"action=\"BuildStopped\" buildProcess=\"msbuild.exe/19804\" eventType=\"BuildEvent\"","thread_name":"Thread-5","@timestamp":"2025-05-12T13:10:35.354+0100","level":"INFO","logger_name":"root"}
```


Figure 32: Positive Results for test No.1,3,4,5

```
build_processes = MSBuild.exe, make.exe, brave.exe
dirs = D:/Documents/Test/Source
fileTypes = cs, cpp
```

Figure 33: Secure Build Config File for test No.2

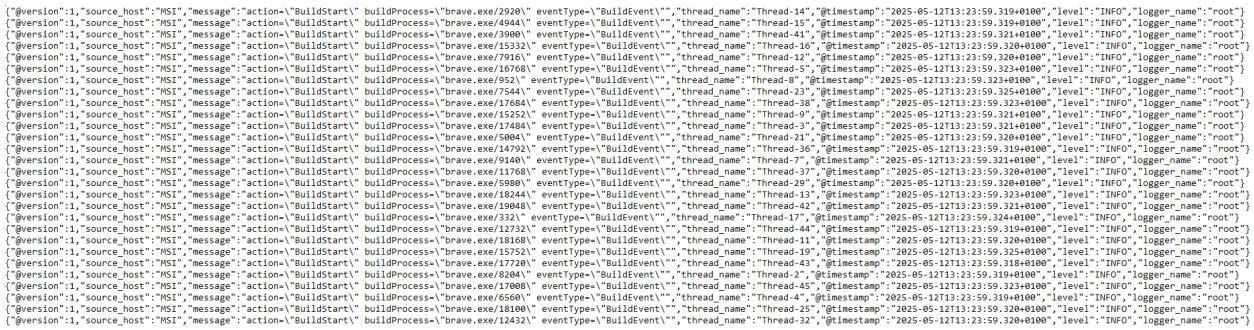

Figure 34: Positive result for test No.2

3.3.2 Testing File Alteration Detection

No.	Test	Description	Pass result	Pass y/ n
1	Detecting File Changes in Configured directory During Build	Configure tool to watch for msbuild processes and changes to the d:/test/source directory then alter a file in that directory whilst msbuild is running	Logs should contain the file altered	y
2	Detecting file changes within sub directories During Build	Configure tool to watch for msbuild processes and changes to the d:/test/source directory then alter a file in a sub folder of the directory whilst msbuild is running	Logs should contain the file altered	y
3	Not Detecting changes to Files outside configured directory	Whilst running msbuild alter files outside the directory that has been configured for monitoring	Logs should not contain any events with altered file in	y
4	Not detecting changes to files when no build process is running	Alter a file in the watched directory when a build process is not running	Logs should not contain changes made outside of build time	y
5	Detecting creation of new source code files during Build	Run msbuild and add new file to the watched directory whilst the build process is running	Logs should show the new file as a file that was altered at build time	y

```
cd D:\Documents\Test\Source
START msbuild .
echo inject >> HelloWorld.cs

Start-Sleep 8
cd D:\Documents\Test\Source
START msbuild .
echo inject >> subfolder/subTest.cs

Start-Sleep 8
START msbuild .
echo "e" >> newFile.cs
```

Figure 35: Power shell script for test No.1,2,5

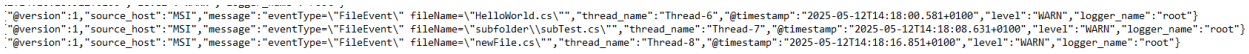

Figure 36: Positive result for No.1,2,5

```
START msbuild .
echo "e" >> ../filename.txt
```

Figure 37: Power shell script for test 3, results in nothing be logged

```
cd D:\Documents\Test\Source
echo "e" >> filename.txt
```

Figure 38: Power shell script for test 4, results in nothing be logged

3.3.3 Testing User Alerts

No.	Test	Description	Pass result	Pass y/ n
1	User is alerted to changes to files of the configured types	Run msbuild process and alter file of one of the configured filetypes repeat for different file types	Logs should contain the file altered	y
2	User is not alerted to alterations of non configured file types	Run msbuild process and alter file that is not a file type found in the config	Logs should contain the file altered	y
3	Correct file name in alert	Run msbuild alter a file during build and check to see if the file altered matches the filename shown to user	Logs should not contain any events with altered file in	y

```
cd D:\Documents\Test\Source
START msbuild .
echo inject >> HelloWorld.cs


Start-Sleep 8
START msbuild .
echo inject >> HelloWorld.txt
```

Figure 39: Power shell script for test No.1,2,3

```
build_processes = MSBuild.exe, make.exe
dirs = D:/Documents/Test/Source
fileTypes = cs
```

Figure 40: Configuration file for test No.1,2,3

Figure 41: User Alerted with correct filename user not alerted for .txt

4 Conclusion

Overall the tool provides the necessary features to detect and warn the user of source file alterations during build time. Additionally it provides SIEM integration making it usable for multi endpoint environments.

The sections above show that whilst developing the tool security best practices where followed this includes sufficient handling of errors, ensuring that dependencies are not vulnerable, using final modifiers to ensure data of events cant be altered and closing system resources after use.

References

[1] Fortinet (n.d.). SolarWinds Supply Chain Attack. [online] Fortinet. Available at: https://www.fortinet.com/resources/cyberglossary/solarwinds-cyber-attack [Accessed 4 May 2025].

[2] Corsetti, S. (2021). Solarwinds Hack: The Intersection of Cybersecurity and Third-Party Risk. [online] ProcessUnity. Available at: https://www.processunity.com/resources/blogs/solarwinds-hack-cybersecurity-third-party-risk/ [Accessed 4 May 2025].

[3] Sylvester, J. (2024). Two Years Later: An Analysis of SolarWinds and the Impact on the Cyber Insurance Industry. [online] Ajg.com. Available at: https://www.ajg.com/news-and-insights/two-years-later-an-analysis-of-solarwinds-and-the-impact-on-the-cyber-insurance-industry/ [Accessed 4 May 2025].

[4] CrowdStrike Intelligence Team (2024). SUNSPOT Malware: A Technical Analysis — CrowdStrike. [online] Crowdstrike.com. Available at: https://www.crowdstrike.com/en-us/blog/sunspot-malware-technical-analysis/ [Accessed 4 May 2025].

[5] microsoft (2024). RTL_USER_PROCESS_PARAMETERS (winternl.h) - Win32 apps. [online] Microsoft.com. Available at: https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-rtl_user_process_parameters [Accessed 5 May 2025].

[6] Neubauer, T. (2023). The what, why and how of event-driven programming. [online] Quix.io. Available at: https://quix.io/blog/what-why-how-of-event-driven-programming [Accessed 6 May 2025].

[7] Oracle (n.d.). Java Security Overview. [online] Oracle Help Center. Available at: https://docs.oracle.com/en/java/javase/11/security/java-security-overview1.html#GUID-65C96219-B2AB-4205-808E-5B41CB2AD694 [Accessed 6 May 2025].

5 Appendix

5.1 Installation Instructions

The file provided is a fat jar and therefore contains all its dependencies all that is needed to run it is Java, to run just execute the power shell script from an admin terminal. For logstash integration alter the setting in log4j2.xml so the tcp appender contains the IP and port of the target logstash server.