TESEC: Accurate Server-side Attack Investigation for Web Applications

Ruihua Wang*†, Yihao Peng*†, Yilun Sun*, Xuancheng Zhang*, Hai Wan* and Xibin Zhao*‡
*KLISS, TNList, School of Software, Tsinghua University
{wrh20,pengyh19,syl21,zxc19}@mails.tsinghua.edu.cn,
{wanhai,zxb}@tsinghua.edu.cn

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179402

***Abstract*—The user interface (UI) of web applications is usually the entry point of web attacks against enterprises and organizations. Finding the UI elements utilized by the intruders is of great importance both for attack interception and web application fixing. Current attack investigation methods targeting web UI either provide rough analysis results or have poor performance in high concurrency scenarios, which leads to heavy manual analysis work. In this paper, we propose TESEC, an accurate attack investigation method for web UI applications. TESEC makes use of two kinds of correlations. The first one, built from annotated audit log partitioned by PID/TID and delimiter-logs, captures the correspondence between audit log entries and web requests. The second one, modeled by an Aho-Corasick automaton built during system testing period, captures the correspondence between requests and the UI elements/events. Leveraging these two correlations, TESEC can accurately and automatically locate the UI elements/events (i.e., the root cause of the alarm) from an alarm, even in high concurrency scenarios. Furthermore, TESEC only needs to be deployed in the server and does not need to collect logs from the client-side browsers. We evaluate TESEC on 12 web applications. The experimental results show that the matching accuracy between UI events/elements and the alarm is above 99.6%. And security analysts only need to check no more than 2 UI elements on average for each individual forensics analysis. The maximum overhead of average response time and audit log space overhead are low (4.3% and 4.6% respectively).**

***Index Terms*—attack investigation, web application forensics, Aho-Corasick automaton, log partitioning**

I. INTRODUCTION

For enterprises and organizations, various web services are often exposed to the outside world, and vulnerabilities in these services are usually potential entry points for security attacks [20], [32]. Web applications generally use a browser-server architecture (B-S architecture) [17]. Users interact with web pages with a browser, which sends requests to the server and these requests will be handled by the web server to provide diverse functions [44]. This interaction model gives intruders opportunities to launch attacks by sending sophisticatedly designed request sequence (such as malicious file uploading, SQL injection, cross-site scripting attacks, etc.). When an attack is detected and alerted by the intrusion detection system (IDS), security analysts hope to find the precise attack entry point (i.e., what request does the intruder send and what UI element is clicked or used to launch the attack) to accurately identify and fix the vulnerability of the web application. However, existing attack investigation methods cannot achieve this goal. The main reasons are as follows.

1) The information bridging attacks and UI elements is missing. Web applications are often composed of front-end UI and back-end business logic. Intruders use vulnerabilities to launch attacks. If the intruder interacts with a UI element to launch an attack, then the UI element is the precise entry point of the attack. Currently when an attack occurs, what can be provided to security analysts are simply the processes related to the attack and the time when the attack occurs. There is no additional information helping linking the attack and the UI element. As a result, security analysts have to spend a lot of time diving into logs to find out the corresponding vulnerability of the web application. Security analysts prefer to be told which UI element and what interaction caused the intrusion, such that they can fix the problem accurately and quickly.

2) Client-side information acquisition is hard. Intuitively, we need additional information describing user behavior from the client PC, if we want to trace back to the UI element. However, for web application with B-S architecture, client-side solutions raise serious security concerns. Firstly, it is impossible to run information collecting software and forensics algorithm in the client PC, since it belongs to an intruder. Secondly, we can not trust the information collected from the client PC for the same reason – the information may be manipulated.

3) Server side applications have high concurrency. Web applications often have high concurrency, since it needs to respond to a huge amount of requests in a short period of time. Under this circumstance, the audit log entries generated by different processes/threads/coroutines handling different requests are often interleaved. Especially, for web frameworks that use coroutines to handle requests, coroutine-switchings, which cannot be perceived by neither audit logs and application logs, will make the logs more confusing and hard to follow. Besides, many log entries always would have the same timestamp, which makes timestamps matching based forensics methods infeasible.

Previous solutions and their problems. For the problem of accurate attack investigation and auditing of web services, Bates et al. proposed NPF [5], which can accurately trace the root cause of common web attacks such as SQL injection and

† Both authors contributed equally to the paper.
‡ Corresponding author.

DOI 10.1109/SP46215.2023.00065

ImageTragick. However, it doesn't record enough information to trace to the exact UI elements/event which causes the attack. UIScope [47] is one of the few solutions that can accurately trace to UI elements using non-instrumentation methods in recent years. UIScope performs causality analysis on both UI elements/events and low-level system events, and then performs the initial event based correlation and timestamp based correlation between system events and UI events. UIScope runs on the PC side. For our goal, we want to find a server-side solution. In other words, **the forensics analysis algorithm only needs to be deployed in the web server and there is no need to collect PC side logs to facilitate the investigation**. UIScope cannot fulfil our goal directly, mainly because: (1) web applications on the server often receives a large number of parallel requests, the timestamp based correlation method cannot solve the log interleaving problem; (2) If the intruder sets the malicious script to sleep for a period of time before attacking, the accuracy of the timestamp based correlation method will significantly drop.

Our solution. Our goal is to propose an attack forensics method for web services, which (1) can accurately find the specific UI element/event that causes the alarm from a given alarm, even in the scenario of log interleaving caused by simultaneous requests; (2) only needs to be deployed in the server and does not need PC side logs.

In the paper, we propose TESEC[1]. In order to find the causal path from the alarm to the UI element/event, TESEC performs the following three steps:

1) Training phase. Each UI event generates a series of network requests. We record the correspondence between UI elements and request sequences in this training phase. System testing is leveraged to get these correspondence relations, since web applications must undergo rigorous tests before officially released. System testing includes simulating user clicks (carried out by automated testing tools, such as `Selenium` [41]) to cover all functions provided by the web service. Then, we can store the correspondence relations between UI elements and request sequences in a database. In order to speed up the processing speed, we adopt Aho-Corasick automaton (hereinafter referred to as AC automaton) to model these relationships.

2) Online running phase. By recording a user's state in cookie, it is possible to distinguish the visits to the web application from different users – different users belong to different sessions. During the online running phase, we attach an AC automaton to each session. Once a web server receives a request, in addition to the original audit log of the operating system, we add an extra log entry called the delimiter log, which contains the session ID, process ID, thread ID, timestamp, and the current state of the attached AC automaton. This delimiter will help us partition the existing audit log, thereby mitigating the log interleaving problem. At the same time, when a request is received, its corresponding session's current state of the attached AC automaton is updated accordingly.

3) Attack investigation phase. When receiving an alarm from the IDS, we first construct a system-level provenance graph from the audit log, and then find the suspicious web server related process starting from the alarm in the provenance graph. After that, we find the delimiter log whose timestamp is closest to the alarm. Then the request sequence, the session ID and the corresponding AC automaton node can be obtained naturally. Finally, we get the set of all possible request sequences and the corresponding set of UI elements/events using the AC automaton and the correspondence relations database.

Experiments and evaluations. We evaluate TESEC on 12 web applications for performance testing. The experimental results show that the matching accuracy between UI elements/events and the alarm entities is above 99.6% in the case of concurrent requests from 500 threads. We achieve an average TOP-2 recall of 93.75% and a TOP-4 recall of 100%, and TESEC find no more than 2 UI elements for each attack averagely, which means that the security analysts only need to check at most 4 UI elements and no more than 2 UI elements on average for forensics analysis. The extra overhead introduced by TESEC is low. In the case of high concurrency of requests, the maximum overhead of average response time and average volume of audit log caused by deploying TESEC is 4.3% and 4.6% respectively. TESEC accurately and efficiently solves the problem of attack investigation in server-side at UI level of web application with low overhead.

Our contributions:

1) We propose TESEC, an accurate attack investigation method for web UI applications. TESEC is able to trace from an alarm event to specific web UI elements/events automatically. TESEC only needs to be deployed on the web server and do not need to collect any logs on the PC side.
2) We design a novel audit log partition technique under high concurrency to precisely match system calls and web requests, and use AC automaton to effectively model the relationship between UI elements and web requests. These techniques provide both accuracy and visibility for web attack investigation.
3) Our solution is general. We implement TESEC for mainstream web frameworks such as Tomcat/Java, Express.js/Node.js and Asyncio/Python. Furthermore, we have already made our implementation open source[2].
4) To evaluate our method, we perform TESEC on 12 web applications, including 2 famous open-source projects and 5 classic vulnerability environments. TESEC achieves 99.6% average accuracy under 500 threads with no more than 5% runtime overhead and space overhead, which shows that TESEC effectively traces the web attacks with visible and accurate results.

[1]TESEC means test+security, because we make full use of the test cases of the web application to build a security model.

[2]https://github.com/tesec-open/tesec

II. Background and Motivation

A. Provenance Graph and Forensics Analysis

A provenance graph is a directed graph that represents the relationship between the subject (process, thread, etc.) and the object (file, registry, network socket, etc.) in the system, where the direction of the edge represents the data flow's direction [29]. We can record the event e that occurs from entity u to entity v at time t as (u, v, t), where e is an edge in the provenance graph. If event $e_1 = (u_1, v_1, t_1)$ and event $e_2 = (u_2, v_2, t_2)$ satisfy $v_1 = u_2$ and $t_1 < t_2$, then we say that e_1 and e_2 have a causal relationship [46]. Forensics analysis [24], [50], [51] is an algorithm that analyzes the origin of an attack and its impact based on the causal relationship in the provenance graph. It includes two steps: backward analysis and forward analysis [18]. Backward analysis can find the origin of the attack, which starts from a symptom event and traces back to obtain a causal path based on the timestamp of the event. The forward analysis algorithm can find a series of events affected by the attack, and it often uses the nodes found by the backward analysis as the entry points.

B. Threat Model

TeSec focuses on attack investigation on the server-side to protect the enterprises or organizations from web attacks. Our attack scenario is that, the back-end of web application (such as a website) is deployed on a server, and attackers intrude the server by exploiting some vulnerabilities in the website. During the process, vulnerabilities such as arbitrary file uploading, arbitrary code execution, and SQL injection are widely exploited by attackers. The entrances of these kinds of attacks are the normal pages provided by the website, that is, benign users can normally access these functions through the browser, while hackers can also use these vulnerable interfaces to launch attacks.

Note that the victim of our attack scenario is the owner of web applications (mostly enterprises or organizations). Attacks to PCs are not considered in our threat model nor attack scenario, for example, using water-hole attack or phishing skills to damage a PC.

We assume that the underlying OS has built-in audit mechanisms that generates audit logs, which should record PID/TIDs[3], and TeSec can insert custom log entries (e.g. Procmon for Windows, and auditd for Linux). We assume that application logs, audit logs, and operating systems are part of the trusted computing base. We also assume that the code of the TeSec system we deploy will not be tampered with by attackers. These assumptions are commonly used in log-based attack investigation methods, such as UIScope [47], Alchemist [48], etc., and there are some common approaches [40] [6] [36] to ensure them.

[3]Linux auditd does not record TIDs by default, so we apply a modification to support it. The details are discussed in Appendix A.

C. Motivating Example

Alice is a security expert for a large enterprise that has more than 10 websites. One day, Alice received an alarm from the IDS and found that a malicious file appeared on the Intranet. Alice quickly began to trace the source of the attack. The traditional analysis procedure is shown in the upper part of Figure 1. By observing the provenance graph generated by the system log, she traced to a process with $pid = 279$ from the read and write operations of the malicious file. She found that 279 was the PID number of *web1.com* belonging to the enterprise. But when she wanted to continue to trace back to the source, she encountered difficulties. Because process 279 connected too many web requests and IPs, she couldn't determine through what IP and which web request of *web1.com* the intruder uploaded this malicious file (as shown in Figure 1, **Difficulty 1: how to locate the suspicious web request from the audit log of malicious events**). So she found Bob, the person in charge of the website, who urgently took the website offline and started checking the web request records of the website. After a lot of works, Bob roughly located 5 file upload points that may cause the attack. He immediately tested these file upload points, and finally found that the intruder leveraged the avatar uploading function and uploaded a malicious script (as shown in Figure 1, **Difficulty 2: how to locate the suspicious web UI element from the suspicious web requests**). Several days had passed when Bob fixed the bug and re-launched the website, and the downtime of the website had brought great losses to the company.

TeSec aims to provide a fast and accurate tracing method for locating web application vulnerabilities. It can start from the provenance graph constructed by audit logs, trace the source to the related web request, and find out what UI element/event caused the alarm. In the above example, if Alice uses TeSec, she can immediately inform Bob that it is the avatar upload function of *web1.com* that caused the problem, so that Bob can focus on troubleshooting this function and quickly fix the vulnerability, thereby reducing the number of downtime caused by the vulnerability and avoiding the loss to the company (as shown in the lower part of Figure 1).

III. System Design

A. System Design Overview

The workflow of TeSec is divided into three phases (shown in Figure 2):

1) Training phase. In the training phase, TeSec aims to establish the relationship between the UI events and the sequences of web requests they generates. To achieve this goal, we collect integrated test cases for web applications, which are written in common testing frameworks such as `Selenium`. During the execution of these test cases, whenever we perform a UI event, we actively add a separator-log entry to distinguish the request sequence generated by the UI event and obtain annotated requests logs. In order to facilitate pattern matching during forensics analysis, we use AC automaton to model the relationship between UI events and request sequences, and

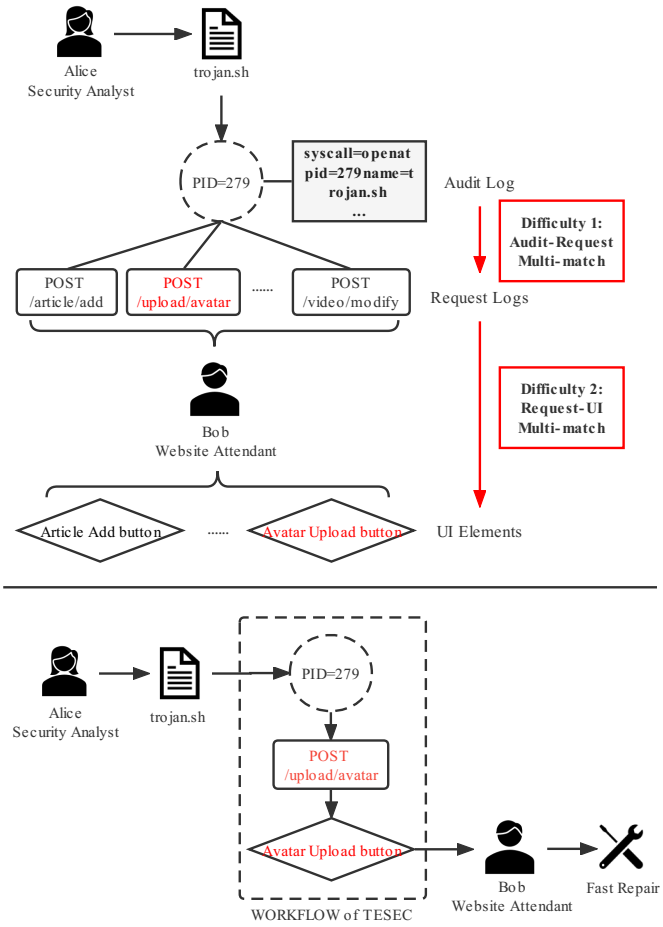

Fig. 1. Motivating example.

store the matching relationships in the *AC Automaton based Relation DB*.

2) Online running phase. To use the service provided by a web application, a user will interact with a web browser and perform real browsing activities, so that the server will receive requests sent by the browser. During the online running phase, for each user, TESEC attaches an AC automaton to its session. When receiving a request, TESEC will insert an extra log entry (i.e., delimiter log) into the audit log of the OS, which contains session ID, process ID, thread ID, timestamp, and the current state of the attached AC automaton. This log entry will help TESEC to partition the existing audit logs. After inserting these delimiter-logs, TESEC obtains the *Annotated Audit Log*. For each session, TESEC will online maintain its node state of the attached AC automaton through the Proxy/Middleware deployed on the server.

3) Attack investigation phase. When an alarm event is detected by the IDS, TESEC aims trace to the specific request and match the possible UI events. Attack investigation phase thus consists of two parts, which are *Event-Request Investigation* and *Request-UI Investigation*. In the first part, using *Malicious Request Analyze*, TESEC analyzes the *Annotated Audit Log* obtained in online running phase, and find the delimiter log (inserted in the online running phase) whose timestamp is closest to the Alarm Event. TESEC records its related information, including the session ID and the corresponding AC automaton node. In the second part, from this *Malicious Request* and its node state in AC automaton, TESEC finds all possible web request sequences. After obtaining these sequences, TESEC finally locates the *Possible List of UI Elements* by querying the *AC Automaton based Relation DB*.

In the workflow of TESEC, there are two key problems (also illustrated in the motivating example):

1) How to find the corresponding web request from a suspicious system call in the audit log?
2) How to locate the UI elements/events from the web request found in the first problem?

We describe our method in detail in Section III-B and Section III-C respectively.

B. Finding the Network Request from an Audit Log Entry

As mentioned in Section I, there are two difficulties with this problem: (1) The information bridging attacks and UI elements is missing and (2) log interleaving caused by multi-process/thread/coroutine mechanism[4].

For the first difficulty, we insert a custom entry in the audit log, which records the identifier of the backend request (we will explain the details of the identifier in III-C). We call this log entry **a delimiter-log**. For the second one, after studying common concurrency mechanism of web applications, we propose a method that can accurately divide the audit log entries corresponding to different requests under high concurrency, so as to achieve the accurate matching between backend requests and audit log entries.

There are currently two main concurrency mechanisms: multi-process/thread based and coroutine based, which are implemented in kernel mode and user mode respectively.

1) Multi-process/thread: For a web application that uses a process pool or a thread pool to handle requests, each process/thread in the pool will sequentially process some requests. When a process/thread finishes processing a request, it will process the next request sequentially. For these requests processed by the same process/thread, the syscalls they trigger will be recorded in audit log sequentially. Hence, we can use the PID and TID to partition the interleaved audit log.

To introduce high-level information, we add a middleware to the web application, so that each worker process/thread would insert a delimiter-log to audit log before starting to process the request. Therefore, when tracing from an audit log entry, we can find the nearest delimiter-log with the same PID/TID, and then the high-level information about the backend request can be acquired.

Since we partition the audit log (including the delimiter-logs) entries based on PID/TID, it does not matter how the

[4]Figure 3 is an example of Linux auditd logs generated when 4 requests are received. As can be seen, although requests 1∼4 were received in sequence, the syscalls triggered when processing them are interleaved. Therefore, we cannot use a simple timestamp comparison method to find the network request from an audit log entry.

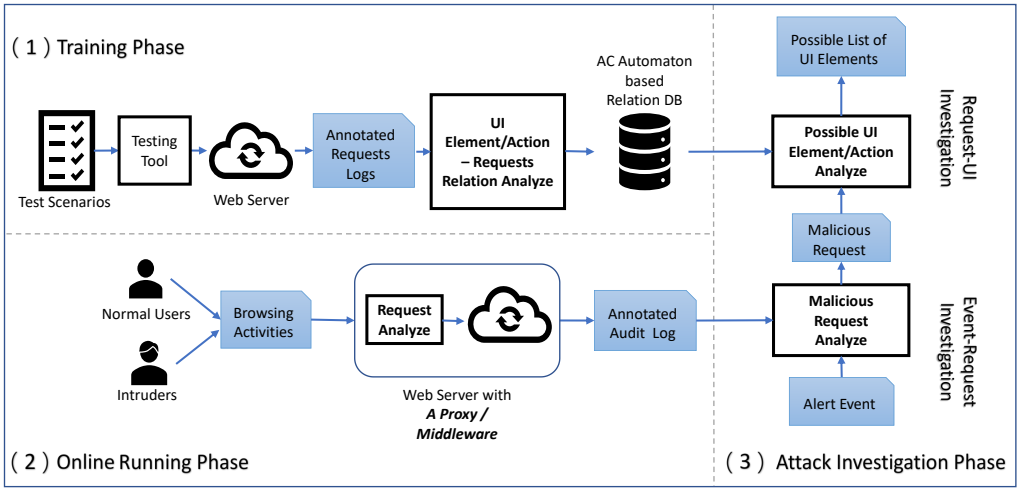

Fig. 2. Overall workflow of TESEC.

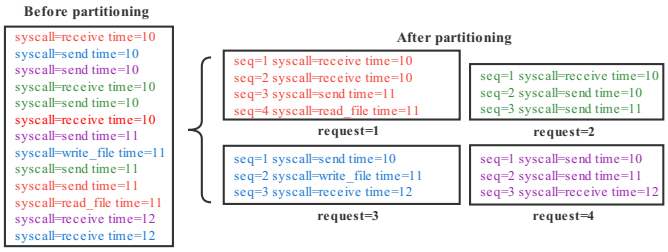

Fig. 3. Interleaving of logs under high concurrency.

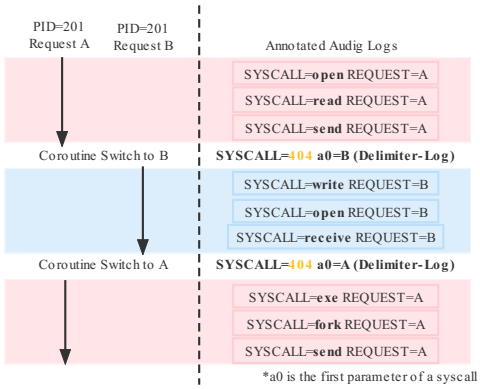

Fig. 4. Delimiter-logs Inserted when Coroutine Switching Occurs.

audit log entries from different processes/threads interleave with each other. The order of audit log entries from the same process/thread is ensured by the underlying OS audit mechanism.

Our modification to the web application is very lightweight. For multiple widely-used backend frameworks, it works like a plugin and there is no need to modify the source code. For implementation details, please refer to Section IV.

Essentially, we use the PID/TID recorded in the audit log to divide multiple requests under high concurrency, which allows us to avoid actively tracking process/thread switching. However, for the coroutine mechanism, we do not have this tricky way.

2) Coroutine: Coroutine is similar to lightweight threads, allowing execution to be suspended and resumed. It has the characteristics of less scheduling and low system overhead. With coroutine mechanism, a web application process/thread may switch multiple times when processing requests, which allows it to handle incoming requests not one by one, but to process multiple requests concurrently in a period of time. Since coroutine switching is a user-mode behavior, audit log does not record this action, so there is no mechanism like PID/TID for us to track the coroutines' behavior.

As a kernel-mode structure, the scheduling of processes and threads is controlled by the operating system. Because the operating system needs to ensure fairness between different processes and threads, some strategies such as Round-Robin are used to switch continuously. However, as a language-level structure, coroutine has a small switching overhead, and the scheduling of the coroutine is often non-preemptive. Coroutine switching occurs only when the coroutine encounters a blocking and voluntarily gives up the right to execute. So it is acceptable for us to actively track the coroutine switching.

As shown in Figure 4, TESEC instrumented the web application to make it inserts a delimiter-log every time a coroutine switching occurs. After partitioning audit log based on PID, considering the audit log entries with the same PID, with the help of the delimiter-logs we inserted, it can be seen that the log entries with same background color belong to the same web request. Similarly, our modification to the web application is very lightweight. Many widely-used backend frameworks with coroutine mechanism are based on interpreted language, so there are some tricks can be used to avoid modifying the original code. For the specific implementation, please refer to Section IV.

After getting annotated the audit logs (i.e., audit logs after inserting delimiter-logs), TESEC manages to match the

Algorithm 1: Matching Web Request

Input : $logs$ is the array of auditd log.
Output : $requests$ is a key-value map of log and the identifier of its matched request.
Variable: $latestRequest$ - A key-value map of PID and its corresponding request identifier.

```
function MatchRequest(logs)
    for log ∈ logs do
        if log is a delimiter-log then
            latestRequest[log.pid] ← log.identifier
        end
        requests[log] ← latestRequest[log.pid]
    end
    return requests
```

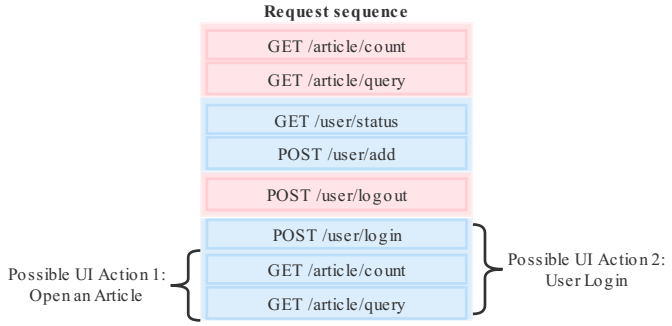

Fig. 5. An example of possible UI elements.

network requests with the audit log entries. For each audit log entry to be traced, we find the nearest delimiter-log before it with the same PID/TID, which records the identifier of a backend request. As shown in Algorithm 1, for the whole audit log, we only need to scan for one time, and maintain the information of the latest delimiter-log of each PID/TID during the scanning process, so that the backend request corresponding to each audit log entry can be determined with linear time complexity.

C. Matching between Network Requests and UI elements/events

Once we get the malicious request, we need to find out which UI element/event triggers this request. For example, as shown in Figure 5, we have identified the malicious request (i.e., the last request "*GET /article/query*") and got the request sequence. However, it is not straight forward to locate the UI elements, since there are possible candidates: the UI event 1 "Open an Article" and the UI event 2 "User Login" both are feasible solutions. The request and UI element/event correlation problem can be stated as follows: Given (1) a set of UI element/event - request sequence pairs $P = \{p|p.element \text{ is a UI element} \wedge p.s \text{ is the request sequence triggerred by } p.element\}$, which models the fact that a UI element/event may generate a request sequence, and (2) for a request sequence S, the set of its all non-empty suffixes: $Suf = \{S[left \cdots L-1] | left \in [0, L-1], \text{where } L \text{ is the length of } S\}$, which models the last few characters that may belong to a UI element/event. We need to find all possible UI elements/events $R \subseteq P$ such that for each $r \in R$, one of the prefixes of $r.s$ is in Suf, in other words, a prefix of $r.s$ is a suffix of S.

This is a classic multi-string matching problem that can be solved using the Aho-Corasick Automaton [1], which is a mature and efficient solution to this problem [25]. AC automaton combines multiple pattern strings to build a tree with returned edges, which is beneficial to reduce the time and space complexity required for string matching [34].

Taking the patterns *a*, *ab*, *bab*, *bc*, *bca*, *c* and *caa* for example, the AC automaton constructed by them is shown in Figure 6 (a). The yellow dotted edge represents the fail pointer. If all the yellow edges are removed, this is a standard Trie. A node colored in blue indicates that a string ends with this node (that is, the path from each blue node to the root node corresponds to a pattern string).

Given a string *cabca*, we want to figure out which patterns are prefixed with one of its suffix. During the running phase shown in Figure 6 (b), AC automaton receives this string character by character. Starting from the root node, if there is a transition edge of the current character, then jump to the corresponding connected node. After two transitions, the current node has no outgoing edge with b, so that it will keep jumping up along the fail edges, until it has a transition edge of this character. So it will reach the node that connected with the root node by a, and then transit along the edge of b. Similarly, it will jump up for one time to reach a node that has outgoing edge with c, and then make a transition. It finally reaches the **START** position in Figure 6 (c).

To locate all the possible patterns, during the query phase as shown in Figure 6 (c), from the last node (node **A**), we need to keep jumping up along the fail edges until the root node, and mark all the nodes passed as well as nodes in their subtrees. The blue solid and marked nodes are all possible patterns (node **A**: *bca*, **B**: *caa*, **C**: *a* and **D**: *ab* in this example). The detailed algorithm is shown in Algorithm 2.

Algorithm 2: Finding Possible Patterns

Input : $node$ is the node that we want to trace from.
Variable: $N.childs$ - A key-value map of characters and child nodes of a node N.
$N.fail$ - Fail pointer of a node N.
$N.mark$ - Is node N marked?

```
function UITrace(node)
    while node ≠ root do
        MarkSubTree(node)
        node ← node.fail
    end
function MarkSubTree(node)
    if node.mark then
        return
    end
    node.mark ← True
    for ch, child ∈ node.childs do
        MarkSubTree(child)
    end
```

In order to make use of AC automaton, we embed AC

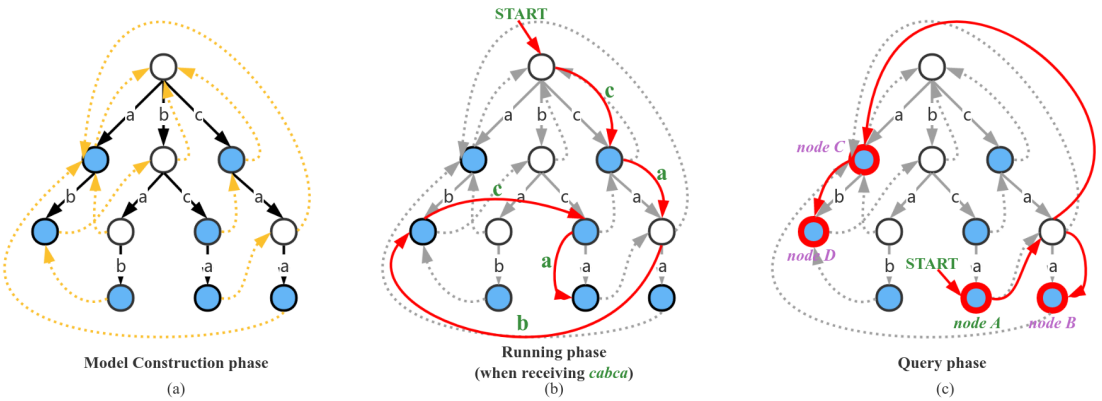

Fig. 6. An example for AC automaton.

automaton into TESEC's 3 phases.

1) AC automaton Construction Phase: During this phase, we first need to get the UI element/event - request sequence pairs, and then construct an AC automaton based on the pairs.

Since a web application that conforms to the software engineering standards needs to be tested sufficiently before it is officially deployed, there are a large number of integration test cases covering all the functions of the web application. We can fully collect and use these test cases to simulate normal UI events, and hence get the UI element/event - request sequence pairs.

Testers generally use automated testing tools like `Selenium` to test the UI of the application. By writing code, testers interact with the UI, simulating real user's usage. Taking `Selenium` as an example, each interactive UI element on the page is an object instance in the test code. When the tester calls the method of the instance, such as "click", the browser will automatically respond accordingly, possibly sending a series of consecutive web requests to the server.

TESEC builds a proxy server for the backend, which will automatically forward the received requests to the real backend server. TESEC will respond to special commands in the header of request, as shown in Table I. When a tester calls a method, a *stop command* is firstly sent to stop recording the backend request corresponding to the previous UI element, and then a *start command* is sent to indicate that a new UI element has started to interact. When all UI elements are tested, a *finish command* is sent to indicate that the test is completed. For example, in our implementation, we only encapsulate the related methods of `Selenium`, and do not need to change the test code, so it can be implemented without instrumentation.

As shown in Figure 7, besides the original requests (in blue and red boxes), the proxy server also inserts several information in between, which stores the UI element/event name and marks the begin and end of the request sequence provoked by the the UI element/event.

Through the above steps, we have obtained the UI element/event - request sequence pairs. Then we construct a AC automaton from these pairs. In the process of construction,

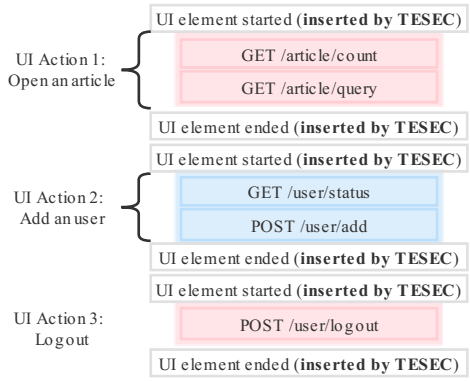

Fig. 7. Workflow for TESEC in model construction phase.

TABLE I
TESEC'S REACTION TO COMMANDS

Command	Reaction
start	Start recording backend requests
stop	Stop recording backend requests
finish	Terminate recording requests and build AC automaton

we treat each request as a character and request sequence as a string, so we follow the standard AC automaton construction steps for construction. The brief construction steps are as follows:

- For all string t_i, where i is a positive integer not exceeding $train\_num$, build a Trie.
- Add a fail pointer to each node in the Trie. For a node i on the Trie, all the edges traversed on the path from the root node to it constitute a string S_i. The fail pointer of node i points to node j if and only if j is the node with the largest length of S_j among all the nodes in the Trie except node i, such that S_j is the suffix of S_i. See Algorithm 3 for details on how to construct fail edges.

The AC automaton can be used for matching and querying. It includes the request sequence corresponding to all normal UI events of web application, and we will use it for subsequent investigation.

Algorithm 3: Building Fail Pointers for AC Automaton

Input : $node$ is the root of the subtree where we want to build fail pointers.
Variable: $N.childs$ - A key-value map of characters and child nodes of a node N.
$N.fail$ - Fail pointer of a node N.

```
function BUILDFAIL(node)
    for ch, child ∈ node.childs do
        failNode ← node
        while failNode ≠ root do
            if ch ∈ failNode.childs then
                child.fail ← failNode.childs[ch]
                break
            end
            failNode ← failNode.fail
        end
        if failNode = root then
            child.fail ← root
        end
        BUILDFAIL(child)
    end
```

2) Running Phase: We attach an AC automaton to each session. During the running phase, we make AC automaton state transition according to the request the web server receives and store the node ID of the current state of the AC automaton. The state transition algorithm of an AC automaton is as follows:

- If the current node has an outgoing edge of the character, then transfer to the corresponding child node.
- If the current node does not have an outgoing edge of the character, then jump up along the fail pointer until there is an outgoing edge of the character, and transfer to the corresponding node.
- If there is still no outgoing edge of the character after jumping to the root node, then transfer to the root node.

For each network request, a transfer on the automaton will occur. Obviously, the worst time complexity is $O(d)$, where d is the depth of the Trie; using an analysis method similar to the KMP algorithm, we can know the average time complexity of each transfer is $O(1)$. And we know that the depth of the Trie can be regarded as a small constant, so it will not have a significant overhead on the system, which can be verified in the experimental part that follows. If a backend request arrives and the node is moved to the root node, it means that the backend request cannot match any UI element, it may be a backend request that does not conform to the UI logic and sent by a script, so we will block this request.

To record the node ID corresponding to each request, we used the delimiter-log inserted in the Section III-B, each of which corresponds to a request. Because we know the automaton node ID corresponding to the request in this section, we can just set the request identifier in the delimiter-log to the corresponding automaton node ID. In this way, we can record the automaton node information of each request in the audit log and system-level provenance graph without adding extra overhead, so as to facilitate subsequent investigation steps.

3) Query Phase: During the query phase, what we have gotten are:

- a request sequence ended with a malicious request
- the AC automaton corresponding to the session of the request
- the current node ID of the AC automaton

Then we can use the query function of AC automaton, such as the example in Figure 6 (c), to get all the possible UI elements/events.

IV. Implementation Details

A. Training Phase

We use a monkey-patch-like[5] approach [39] to change the behavior of some functions in `Selenium`, as a testing tool to execute test code in the training phase. In training phase, the proxy server is implemented in `Flask`.

B. Online Running Phase

1) Audit Log Sources: For Linux we use auditd as the audit log source, while for Windows we use Process Monitor (Procmon).

2) Delimiter-Logs Insertion Mechanisms for Different Operating Systems: The mechanisms to insert delimiter-logs in Linux and Windows are different:

- **Auditd (Linux)**. TESEC calls a non-existent syscall with a fixed number[6] to insert a delimiter-log into the audit log. Since auditd will record the parameters of a syscall, we could record more information such as the node id of the AC automaton in its first parameter (a0).
- **Procmon (Windows)**. TESEC inserts a Profiling Events of operation type "Debug Output Profiling" to insert a delimiter-log into the audit log. Because procmon will record the "Detail" filed of this type of event, we could record the same information as auditd in this field.

Both mechanisms can be easily implemented in C++. For programming languages other than C++, such as Java, Node.js, Python, which are used in our evaluations, since they all have the ability to import C++ code, the above two mechanisms can be implemented naturally.

3) When and How to Invoke the Insertions for Mainstream Web Frameworks: In this part, we show when and how to invoke the above delimiter-log insertion for mainstream web frameworks such as Tomcat/Java, Express.js/Node.js and Asyncio/Python. Since current mainstream web frameworks often provide sufficient interfaces, leveraging which TESEC can invoke the insertion operations in a elegant and plugin-like way without modifying the source codes.

- **Tomcat (Multi-process/thread)**. `Tomcat` [45] is a web server where web applications written in Java can be deployed. We use its "filter" mechanism to invoke the insertion. Each filter has a matching rule. If the request path hits the matching rule of a filter, the thread processing the request will call the filter first, and then execute the

[5]Dynamically replace methods or properties at runtime.
[6]In this paper, we choose 404.

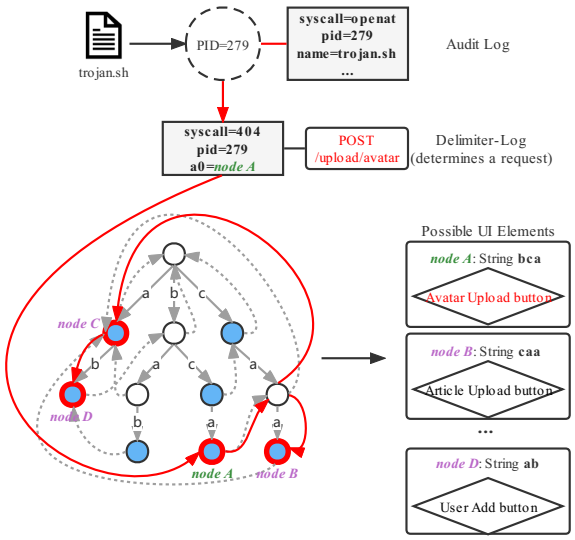

Fig. 8. An example of Attack investigation algorithm.

corresponding processing function. Throughout the process, the TID is constant. Therefore, TESEC could utilize a filter to invoke the insertion.

- **Express.js (Coroutine)**. `Express.js` [7] is based on `Node.js`, which inherently uses coroutines. In fact, `Node.js` provides the library *async_hooks* to facilitate tracking asynchronous events. We only need to filter out the coroutine switching events not related to the request, and then we can use the callback function provided by *async_hooks* to invoke the insertion.
- **Asyncio (Coroutine)**. For `Python`'s `Asyncio` module, we can use its built-in *loop.set_task_factory* function to modify the default behavior of creating coroutine tasks. When a coroutine task is created, TESEC encapsulates the coroutine with a subclass of *asyncio.coroutines.CoroWrapper* for debugging, and overload its *send* method to invoke the insertion.

C. Attack Investigation Phase

We summarize the complete attack investigation steps of TESEC with an example in Figure 8. Starting from an alarm event provided by IDS, the detailed work flow is as follows:

1) We use SPADE to construct provenance graphs from collected server-side audit logs[7]. In the provenance graph, we start from the malicious file and trace back until reaching a process corresponding to the web application (i.e., web server) that caused the uploading of the file. Hence, we get **the malicious audit log**, which is related to the malicious file uploading action.

2) With the help of the precise log partition, TESEC starts from the malicious audit log entry, and finds **the latest delimiter log** inserted by TESEC in the audit logs. From the delimiter log, TESEC further finds **the web request** that is highly relevant to the uploading action of the malicious file.

3) TESEC has already collected test cases covering all the web functions to build the AC automaton, which captures the relation between a UI element/event and its corresponding web request sequence, in the training phase. Once given the web request and the current state of the AC automaton, we can query the AC automaton to find **all the possible request sequences** and **their corresponding UI events** as well.

After the above steps, we have identified all UI elements/events that could lead to this attack. Finally, we inform security experts of these UI events that may be related to the vulnerability, helping them quickly locate and fix the problem.

[7]TESEC uses SPADE [9]–[11] to obtain a system-level provenance graph. SPADE is an open-source provenance graph generation tool, which converts Linux auditd logs or Windows ETW logs into a standard-format provenance graph. Using the provenance graph provided by SPADE, we can trace from the alarmed entity (such as some Trojan files uploaded using web application vulnerabilities) to the corresponding entity of the web server process of the invaded web application. For example in Figure 1, given the malicious node "$trojan.sh$" we can find the process node with $pid = 279$ from the provenance graph.

V. EVALUATION

We designed experiments to answer the following questions:

1) How accurately can TESEC perform entity matching in the case of high concurrency of requests and interleaving of logs? (Section V-B)

2) During an attack investigation, from the suspicious audit log entries, how accurately can TESEC find the corresponding UI elements? (Section V-C)

3) Compared with the timestamps based correlation method, how much improvement does TESEC have in the matching result? (Section V-D)

4) If an attacker deliberately rejects all cookies, how this impacts the accuracy of the investigation ? (Section V-E)

5) What are the runtime and space overhead of TESEC when deployed in real-world environments? (Section V-F)

A. Experiment Setup

Linux server we used for experiments has an Intel Xeon Platinum 8361HC CPU 2.60GHz and 8GB RAM, running Ubuntu 18.04.4. Windows server we used has an Intel Core i7-8565U CPU 1.80GHz and 16GB RAM, running Windows 10. In order to test the performance of TESEC, as shown in Table II and III, we designed 12 web application scenarios, and conducted several experiments in each scenario. In order to obtain the annotated audit log, different implementations are required for different backend frameworks.

We implemented TESEC for the following frameworks:

1) `ASGI` framework (a Gateway Interface in `Python`). It is suitable for backend frameworks that use `Python`'s `Asyncio` module, including `Starlette` [43], `FastAPI` [8] and `Aiohttp` [3]
2) `Tomcat` (a web server for deploying `Java` web applications)
3) `Express.js` (a backend framework in `Node.js`).

The implementation details are described in Section IV.

TABLE II
BASIC INFORMATION OF WEB APPLICATIONS

App Name	Backend Framework	What to Test	OS	Number of Backend APIs
App 1	FastAPI	Investigation effect & Performance	Linux	5
App 2	FastAPI	Investigation effect & Performance	Linux	13
App 3	Express.js	Investigation effect	Linux	42
App 4	Flask	Investigation effect	Linux	24
App 5	Aiohttp	Investigation effect	Linux	11
App 6	Tomcat + Springboot	Investigation effect	Windows	64
App 7	Express.js	Investigation effect	Linux	18

TABLE III
COMMON VULNERABILITY ENVIRONMENT

App name	Weakness	Source	Language
App 8	Deserialization	CVE-2021-44228	Java
App 9	Injection	CVE-2021-21315	Node.js
App 10	XML eXternal Entity	XXE-LAB	Python
App 11	Broken Access Control	CVE-2016-4437	Java
App 12	Type Conversion	CVE-2017-8291	Python

Some basic information about App 1-7 are given in Table II and detailed descriptions are in Appendix B.

App 8-12 are all environments from Proof of Concept (POC) to common vulnerabilities, and can be used to evaluate the accuracy of attack investigation in real environment. Details about these 5 applications are described in Appendix C.

B. The Accuracy of Entity Matching

TESEC contains two kinds of entity matchings. The first one is the matching between audit log entries and backend requests, and the second one is the matching between backend requests and UI elements. We evaluate them separately.

1) Can audit log entries be correctly matched with backend requests?: In this section, we evaluate whether TESEC can correctly match audit log entries with web requests.

We modified App 1 to record the matching information between syscalls and web requests at runtime. App 1 has 3 backend APIs corresponding to file reading, modification and deletion operations respectively with the filename that it operates on as the parameter. For file reading and modification operations, the **open** syscall and the filename will be recorded in the auditd log; for file deleting operations, the **unlink** syscall and the filename will be recorded. Hence, the correspondence between audit log entries (i.e.,**open**, **unlink** syscalls) and web requests can be simply derived using the filename. In this way we get the ground-truth.

We use 10, 100, and 500 threads to concurrently initiate access to App 1, and then evaluate the matching accuracy. Table IV shows the results. Since the web requests and syscalls of App 1 are one-to-one, the accuracy and recall are the same. It can be seen that in the case of high concurrency of requests and interleaving of logs, the matching accuracy is still higher than 99%. The reason why the accuracy is less than 100% is that: auditd may lose audit log entries in the case of high throughput, so that some 404 syscalls cannot be successfully written to auditd, and matching failure occurs. In other words, if there is no loss of auditd log entries, then both the accuracy and recall should be 100%.

TABLE IV
MATCHING RESULT OF BACKEND REQUESTS AND AUDIT LOG ENTRIES (APP 1)

Concurrent Threads Number	Request Number	File Syscall Number	Correct Number	Accuracy	Recall
10	3537	2566	2566	100%	100%
100	35081	25063	25046	99.93%	99.93%
500	289781	122879	122436	99.64%	99.64%

Using the same method as above, we evaluated the matching accuracy between backend requests and audit log entries for App 6 and 7. App 6 is deployed in Windows with Procmon to monitor its behavior, while App 7 is deployed in Linux with auditd. As shown in table V, the matching results for them are perfect as expected.

TABLE V
MATCHING RESULT OF BACKEND REQUESTS AND AUDIT LOG ENTRIES (APP 6,7)

App	Concurrent Threads Number	Syscall Number	Accuracy	Recall
App 6	500	300000	100%	100%
App 7	500	300000	100%	100%

2) Can web requests be correctly matched with UI elements?: TESEC may find several UI elements when matching with web requests. In this section, we use App 2-5 to find out how many UI elements in general that TESEC would finds in real applications.

To evaluate the recall of matching, we let 5 users to use Selenium script to randomly access the website, and record the UI elements used. The experiment lasts for 20 minutes and generates many backend requests. We record the numbers of UI elements traced back from the backend request using our algorithm. The results are shown in Figure 9 and Table VI. For all applications, the average TOP-1 recall is 74%, and the average TOP-2 recall is 93.75%. This means that the number of UI elements is less than 2 in most of the cases (shown in Table VI). In other words, TESEC greatly reduces the analysis scope and workload of security analysts.

TABLE VI
AVERAGE NUMBER OF UI ELEMENTS MATCHED BY THE WEB REQUESTS

Application	App 2	App 3	App 4	App 5
Average number of UI elements	1.46	1.66	1.04	1.08

C. Attack Investigation Accuracy

In this section, we evaluate the accuracy of attack investigation from audit log entries to UI elements.

We planted several vulnerabilities in Apps 2, 4 and 5, and attempted to find the UI elements from syscalls caused by the vulnerabilities. In this experiment, 5 users used Selenium script to randomly visit the website of Apps 2, 4 and 5 normally, and record the interactive UI elements. For each application, the experiment lasted for 20 minutes, while the attacker used the vulnerability to do file operations. We

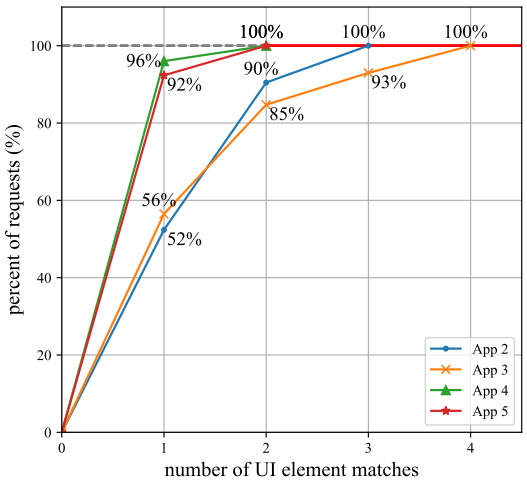

Fig. 9. Cumulative graph of the number of UI elements matched by the web requests.

added the following vulnerabilities: (1) **Arbitrary Command Execution Vulnerability**. The attacker modifies files at specified locations through this vulnerability. (2) **Arbitrary File Upload Vulnerability**. The attacker modifies files at specified locations through this vulnerability. (3) **Arbitrary File Read Vulnerability**. The attacker reads the specific file through this vulnerability.

As shown in Table VII, vulnerability-related syscalls are matched accurately to the vulnerability-related UI element.

In order to evaluate our method for real application with vulnerabilities, we used 5 applications from public POCs to common vulnerabilities. We attacked the applications as described in POCs several times and attempted to find the UI elements from the audit log entries generated by the attack. We refer to Owasp Top-10 2017 [35] to select the types of these vulnerabilities. In a word, for all applications, TESEC can always uniquely and correctly find the corresponding UI elements. The details of each application are described in Appendix C, and Figure 10 contains the provenance graphs of each application.

TABLE VII
COMPLETE INVESTIGATION OF ATTACK

Application	Vulnerability Type	Number of Requests	Accuracy	Recall
App 2	Arbitrary Command Execution	102	100%	100%
	Arbitrary File Upload	100	100%	100%
	Arbitrary File Read	106	100%	100%
App 4	Arbitrary Command Execution	236	100%	100%
	Arbitrary File Upload	221	100%	100%
	Arbitrary File Read	254	100%	100%
App 5	Arbitrary Command Execution	232	100%	100%
	Arbitrary File Upload	243	100%	100%
	Arbitrary File Read	216	100%	100%

D. Comparison with Timestamp Matching-based Method

UIScope uses timestamps to match system-level and UI-level logs. However, in the high concurrency scenario of web services, audit logs will be interleaved, so it is difficult to match them directly by timestamps.

Similar to Section IV.B, we use App 1 to evaluate the accuracy of matching between web requests and UI elements. We designed two scenarios. In one scenario, App 1 sleep for a random time interval while processing each request and in the other scenario App 1 directly process each request without sleeping.

The results are shown in Table VIII. It can be found that in the case of high concurrency of requests, the accuracy of direct timestamp comparison method greatly dropped. Comparing with the results in Table IV, the accuracy and recall of TESEC is still over 99%, which shows that TESEC has a huge advantage compared with timestamp matching-based method.

TABLE VIII
MATCHING RESULT OBTAINED BY DIRECT TIMESTAMP COMPARISON METHOD

	Concurrent Threads Number	File Syscall Number	Correct Number	Accuracy	Recall
Sleep for Random 0-5s per Request	50	6223	1166	18.74%	18.74%
	100	12380	2125	17.16%	17.16%
	500	62348	7973	12.79%	12.79%
No Sleep	50	6057	5714	94.34%	94.34%
	100	12554	10652	84.85%	84.85%
	500	62597	12447	19.88%	19.88%

E. Impact of Cookie Rejection

TESEC uses cookies to distinguish different users and partition their behaviors. If an attacker deliberately rejects the cookie, then the investigation ability of TESEC will be negatively affected. In this case, TESEC is considered to be effective if one of the following two conditions is met:

1) TESEC is able to detect whether the user has rejected cookies. If an attacker rejects all cookies, it will behave like a new user to the web application – since a new user sends requests without any cookies on their first visit to the web application. Then, if there is no corresponding outgoing edge from the root node in AC automaton, it can be inferred that the user deliberately rejects the cookie, and this condition is met.

2) Though TESEC cannot detect the situation that cookies are rejected, it still can successfully find the corresponding UI element. It can be noticed that if a user only sends one backend request when interacting with a UI element, then TESEC will not be able to determine whether the sender has deliberately rejected the cookie, but TESEC can still find the UI element and this condition is met.

If both conditions are not met, TESEC's attack investigation between UI elements and backend requests will be unreliable, but it will not affect the attack investigation between backend requests and audit log entries.

App 3 is used to do the evaluation. We interacted with all UI elements of App 3 with cookies turned off. The results shows that, for 52.38% UI elements, TESEC can tell that the

cookies are turned off (i.e., condition 1 is meet); for 40.48% UI elements, TESEC can trace to that very UI element (i.e., condition 2 is meet); only for 7.14% of UI elements, TESEC neither detected the cookie rejection nor found the correct UI elements.

These backend requests that do not satisfy both conditions are starting requests for some UI elements. For example, when entering a form filling page, request A is sent. When the user fills the form and clicks the "Submit" button, the form will be uploaded and request B will be sent. After that, the page will be refreshed and request A will be sent again. In this case, when the backend receives BA, since the cookie is rejected, the A request will be traced back to the UI element of "entering the page" instead of "clicking the button".

Without recording cookies, if an attacker interacts with different UI elements multiple times, TESEC can detect it with a high probability. At that time, the security analysts can locate the attacker from alerts through IP or other methods.

F. Performance Overhead

1) Runtime Overhead: The deployment of TESEC incurs additional runtime overhead:

- TESEC maintains the transfer on AC automation online when it receives a backend request.
- TESEC generates a 404 syscall every time the coroutine is switched.

We evaluate the influence of TESEC on the system response time. For App 1, we use 10, 100 and 500 threads to concurrently access it, and visit all 5 APIs randomly. For App 2, we use 50 and 100 threads.The results are shown in Table IX. It can be found that for App 1, as the number of concurrent threads continues to increase, the overhead of the average response time does not increase significantly. It remains at about 1%. There is no obvious change in the maximum and minimum response time. For App 2, the average response time overhead remains within 5%, and the maximum and minimum response time do not change significantly.

2) Space Overhead: Space overhead incurred by the deployment of TESEC consists of two parts:

- TESEC records the current automaton node ID in each user's session. Node ID can be recorded in cookie or in a database. Since the node ID is just a number, we consider the space overhead caused by this part to be negligible.
- TESEC inserts delimiter-logs in the audit log, which will cause space overhead.

Therefore, we counted the change of audit log sizes before and after TESEC is deployed.

The evaluation method is the same as that of Runtime overhead. While recording the response time, we also recorded the change of audit log size, as shown in Table X. It can be found that, regardless of the number of concurrent threads, the number of lines in the audit log increased less than 5% in both applications. Considering the different configurations of auditd in different production environments, we recorded the number of 404 syscalls. A 404 syscalls will generate 2 lines of auditd log entries, so averagely a request will generate less than 4 lines of auditd log entries.

Since each coroutine switch will trigger a 404 syscall, for a request, the number of 404 syscall can represent the switching times of the coroutine processing this request. We found that even in the case of high concurrency of requests, the average number of coroutine switching times is low.

VI. LIMITATION AND DISCUSSION

A. Auditd Log Entries Loss at High Throughput

In Evaluation, we mentioned that the auditd log may lose log entries in the case of high throughput, and will not record the 404 syscall triggered by TESEC, thus causing the matching failure. Our idea is to consider saving the log entries of the 404 syscalls as a separate file, and confirm its location in the auditd log through timestamp matching, so as to avoid too high auditd log throughput. However, during the experiment, we found that the timestamp recorded by auditd has a certain deviation from the timing function that comes with `Python`, and it is difficult to match. And the timestamp of auditd is not accurate enough, there will be multiple syscalls with the same timestamp. Therefore, we currently do not have a good solution to this problem.

B. How to Handle Concurrent Backend Requests from the Same Session?

We believe that under normal circumstances, the backend requests generated when interacting with UI elements arrive at the backend server one by one in a fixed order, which is also the premise that we can model backend requests with strings. However, in real world, sometimes the browser calls the backend request asynchronously. At this time, the backend requests arriving at the backend will be in an indeterminate order. If we want to learn the correlations between UI elements of this type and the request sequences they might trigger through TESEC, we may have to enumerate all possible request orders. At this time, it is inevitable to modify the test code. We believe that we can use the idea of automating the analysis of JS code for testing, and automatically analyze the backend requests whose relative order may change, reducing the workload of manual testing of the code.

C. How to Handle Backend Requests that are Sent Periodically?

There is also a special phenomenon in the real world, that is, after interacting with a UI element, the page may be controlled by JS code and send requests to the backend periodically. Such a request may be inserted between consecutive backend requests generated by other UI elements, which interferes with the investigation on the latter. In order to solve this special case, we can add some information to the backend API documentation to let TESEC ignore this kind of backend request.

TABLE IX
COMPARISON OF RUNTIME

App	Threads Number	Request Number		Average Response Time (s)			Maximum Response Time (s)		Minimum Response Time (s)	
		No TESEC	With TESEC	No TESEC	With TESEC	Overhead	No TESEC	With TESEC	No TESEC	With TESEC
App 1	10	3469	3537	2.5077	2.5337	1.0%	5.2050	6.0101	0.0030	0.0031
	100	34826	35081	2.5366	2.5386	0.1%	5.5001	5.5728	0.0028	0.0029
	50	175425	175079	2.6139	2.6403	1.0%	7.9604	6.7906	0.0028	0.0027
App 2	100	20000	20000	0.0092	0.0096	4.3%	0.1500	0.1597	0.0045	0.0048
	500	20000	20000	0.0120	0.0121	0.8%	0.3641	0.2624	0.0046	0.0048

TABLE X
COMPARISON OF SPACE

App	Threads Number	Request Number		Number of auditd Log Lines			Number of 404 Syscalls	Average Number of 404 Syscalls
		Without TESEC	With TESEC	Without TESEC	With TESEC	Overhead	Without TESEC	With TESEC
App 1	10	3469	3537	903060	922451	2.1%	5236	1.61
	100	34826	35081	4973170	5075887	2.1%	55028	1.57
	50	175425	175079	22494240	23080385	2.7%	55028	1.66
App 2	100	20000	20000	1172180	1221896	4.2%	24612	1.23
	500	20000	20000	1085990	1135516	4.6%	25013	1.25

VII. RELATED WORK

A. Attack Investigation Using Provenance Graphs

The provenance graph, proposed by BackTracking [23], [24], expresses the causal relationship between entities in computers. Entities are divided into subjects (processes, threads, etc.) and objects (sockets, files, etc.). Provenance graphs are usually generated using logs, and most articles [14], [30] use audit logs to generate system-level provenance graphs. SPADE [9]–[11] provides a set of general audit-level provenance graph generation tools from multiple operating systems, and graphs can be automatically built by collecting system logs. Camflow [37], [38] provides a kernel-level provenance graph generating tool for Linux. It is based on the LSM module of the Linux kernel, which observes system objects and their communication in the kernel. Camflow [12], [13] realizes a more fine-grained provenance graph generation.

Forensics analysis algorithm [50] [51] [23] [24] [22] [27] [31] [19] [26] [49] can be executed on the provenance graph. The standard forensics analysis algorithm consists of two steps: firstly it performs backward analysis from symptom events, and then it performs forward analysis starting from the entities found in the backward analysis and finds out the possible impact of the attack. The accuracy of forensics analysis algorithm is often affected by dependency explosion problem, which refers to the fact that an entity (often corresponding to a long-running process) may be connected with too many input edges and output edges, such that all output edges of the entity are considered to be dependent on all its input edges.

In order to solve the dependency explosion problem, BEEP imported binary instrumentation method to divide long-running processes into several units. MPI strengthened the effect of execution partition by allowing users to annotate the source code to mark the high-level unit. Although MPI also inserts messages into audit logs, it is quite different from TESEC for the following reasons: (1) MPI takes the source code as input. It first makes some annotations to the source code such that it can weave the instrumentation into the executable. It requires and changes the source code and is then somehow heavyweight, which has two consequences: 1) if the source code is not available, MPI cannot apply; 2) the whole process of MPI should be redone if the software is updated. TESEC leverages the internal features of web frameworks and is implemented in a plug-in fashion. As a result, it is orthogonal to the software updates and lightweight. (2) It is common that web frameworks are implemented in interpreted languages, so there is no compilation process and no intermediate product like LLVM. As a result, the MPI solution cannot apply. (3) For most web applications, they import various third-party libraries. For example, the coroutine scheduling method of `Python` can be imported and modified at will. It is difficult for MPI to automatically and statically calculate the location that needs to be instrumented. So that MPI needs to recalculate for each specific application, while TESEC only needs to instrument a web framework for one time. (4) In order to find the request, additional information should be recorded, which leads to a specific amendment to MPI's annotation and compilation process. This amendment connot be done in a general fashion, in our opinion.

B. Accurate Attack Investigation on Web Applications

Some work uses finite state machines [2] to model web services' front-end to back-end network requests. Haydar et al. proposed [16] for automatic modeling and formal verification of web applications using communicating finite automaton. This method can complete state machine modeling for complex web applications of multi-window/frame. Some articles [33] also proposed methods for detecting and preventing SQL injection attacks using Aho-Corasick automaton. These methods [21] analyze the patterns in SQL statements and use the multi-pattern matching function of AC automata to detect possible attack patterns. NPF [5] proposes a forensics analysis method for web applications using binary instrumentation.

Few articles can solve the problem of UI-level attack investigation. UIScope proposes a solution on PC side. It gives an algorithm for generating the UI-level provenance graph using the logs from Windows UI Automation. It has also designed timestamp matching strategies to merge the system-

level provenance graph and the UI-level provenance graph. UIScope's UI-level solution can hardly migrate from PC-side to server-side and be used on web applications, because logs recording the interactions between users and browsers can not be collected on server-side, and UIScope's timestamp matching strategy can not handle high concurrent cases. There exists some client-side forensics researches and products, such as Mnemosyne [4], JsGraph [28] and Zipkin [52], which can also be used for web attacks provenance.

VIII. Conclusion

We propose TeSec, a UI-level accurate attack investigation algorithm for web applications. TeSec uses log partition technology to achieve accurate provenance from system calls to network requests, and uses AC automaton to achieve accurate matching from network requests to UI elements. When an alarm occurs, TeSec can start from the symptom event and trace back to the UI event that may cause that event, thereby helping security analysts quickly fix the vulnerabilities. We use 12 applications to evaluate the effect of TeSec. The experimental results show that TeSec achieved an average accuracy of more than 99.6% in the case of high concurrency of requests with only 4.3% runtime overhead and 4.6% space overhead. We implements TeSec for several web frameworks in Python, Java and Node.js. We have already made our implementations open source.

Acknowledgment

This work was supported in part by Key-Area Research and Development Program of Guangdong Province under Grant 2020B010164001, NSFC Program (No. 62076146, U20A6003, 62021002, U1801263, U19A2062, U1911401, 62127803).

References

[1] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[3] Aiohttp. *https://docs.aiohttp.org/en/stable/*.

[4] Joey Allen, Zheng Yang, Matthew Landen, Raghav Bhat, Harsh Grover, Andrew Chang, Yang Ji, Roberto Perdisci, and Wenke Lee. Mnemosyne: An effective and efficient postmortem watering hole attack investigation system. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 787–802. ACM, 2020.

[5] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *Proceedings of the 26th International Conference on World Wide Web*, pages 887–895, 2017.

[6] Adam Bates, Dave (Jing) Tian, Kevin R.B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. *24th USENIX Security Symposium*, 2015.

[7] Express. *https://expressjs.com/*.

[8] FastAPI. *https://fastapi.tiangolo.com/*.

[9] Ashish Gehani, Raza Ahmad, Hassaan Irshad, Jianqiao Zhu, and Jignesh M. Patel. Digging into big provenance (with SPADE). *Commun. ACM*, 64(12):48–56, 2021.

[10] Ashish Gehani, Hasanat Kazmi, and Hassaan Irshad. Scaling SPADE to "big provenance". In Sarah Cohen Boulakia, editor, *8th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2016, Washington, D.C., USA, June 8-9, 2016*. USENIX Association, 2016.

[11] Ashish Gehani and Dawood Tariq. SPADE: support for provenance auditing in distributed environments. In Priya Narasimhan and Peter Triantafillou, editors, *Middleware 2012 - ACM/IFIP/USENIX 13th International Middleware Conference, Montreal, QC, Canada, December 3-7, 2012. Proceedings*, volume 7662 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2012.

[12] Xueyuan Han, James Mickens, Ashish Gehani, Margo I. Seltzer, and Thomas F. J.-M. Pasquier. Xanthus: Push-button orchestration of host provenance data collection. In Ivo Jimenez, Carlos Maltzahn, and Jay F. Lofstead, editors, *Proceedings of the 3rd International Workshop on Practical Reproducible Evaluation of Computer Systems, P-RECS@HPDC 2020, Stockholm, Sweden, June 23, 2020*, pages 27–32. ACM, 2020.

[13] Xueyuan Han, Thomas F. J.-M. Pasquier, Adam Bates, James Mickens, and Margo I. Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[14] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. *NDSS*, 2019.

[15] Wajih Ul Hassan, Mohammad Ali Noureddine, Pubali Datta, and Adam Bates. Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In *Network and Distributed System Security Symposium*, 2020.

[16] May Haydar, Alexandre Petrenko, and Houari A. Sahraoui. Formal verification of web applications modeled by communicating automata. In David de Frutos-Escrig and Manuel Núñez, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2004, 24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004, Proceedings*, volume 3235 of *Lecture Notes in Computer Science*, pages 115–132. Springer, 2004.

[17] Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. Next stop, the cloud: understanding modern web service deployment in EC2 and azure. In Konstantina Papagiannaki, P. Krishna Gummadi, and Craig Partridge, editors, *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013*, pages 177–190. ACM, 2013.

[18] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[19] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. {SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data. *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017.

[20] Meiko Jensen, Nils Gruschka, and Ralph Herkenhöner. A survey of attacks on web services. *Comput. Sci. Res. Dev.*, 24(4):185–197, 2009.

[21] Swapnil Kharche, Kanchan Gohad, Bharti Ambetkar, et al. Preventing sql injection attack using pattern matching algorithm. *arXiv preprint arXiv:1504.06920*, 2015.

[22] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[23] Samuel T King and Peter M Chen. Backtracking intrusions. *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[24] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. Enriching intrusion alerts through multi-host causality. *NDSS*, 2005.

[25] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[26] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. Mci: Modeling-based causality inference in audit logging for attack investigation. *NDSS*, 2018.

[27] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. *NDSS*, 2013.

[28] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[29] Zhenyuan Li, Qi Alfred Chen, Runqing Yang, Yan Chen, and Wei Ruan. Threat detection and investigation with system-level provenance graphs: A survey. *Comput. Secur.*, 106:102282, 2021.

[30] S. Ma, X. Zhang, and D. Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. *NDSS*, 2016.

[31] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for windows. *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015.

[32] Nicolás Montés, Gustavo Betarte, Rodrigo Martínez, and Álvaro Pardo. Web application attacks detection using deep learning. In João Manuel R. S. Tavares, João Paulo Papa, and Manuel González Hidalgo, editors, *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications - 25th Iberoamerican Congress, CIARP 2021, Porto, Portugal, May 10-13, 2021, Revised Selected Papers*, volume 12702 of *Lecture Notes in Computer Science*, pages 227–236. Springer, 2021.

[33] Danapaquiame Nagamouttou, Ilavarasan Egambaram, Muthumanickam Krishnan, and Poonkuzhali Narasingam. A verification strategy for web services composition using enhanced stacked automata model. *SpringerPlus*, 4(1):1–13, 2015.

[34] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[35] OWASP. *https://owasp.org/www-project-top-ten/2017/Top_10*.

[36] Thomas F. J.-M. Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David M. Eyers, Margo I. Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 405–418. ACM, 2017.

[37] Thomas F. J.-M. Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David M. Eyers, Jean Bacon, and Margo I. Seltzer. Runtime analysis of whole-system provenance. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1601–1616. ACM, 2018.

[38] Thomas F. J.-M. Pasquier, Jatinder Singh, David M. Eyers, and Jean Bacon. Camflow: Managed data-sharing for cloud services. *IEEE Trans. Cloud Comput.*, 5(3):472–484, 2017.

[39] Monkey Patch. *https://en.wikipedia.org/wiki/Monkey_patch*.

[40] Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-fi: Collecting high-fidelity whole-system provenance. *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[41] Selenium. *https://www.selenium.dev/*.

[42] SQLAlchemy. *https://www.sqlalchemy.org/*.

[43] Starlette. *https://www.starlette.io/*.

[44] Robert Tolksdorf. Web servers, clients, and browsers. In Richard Zurawski, editor, *The Industrial Information Technology Handbook*, pages 1–8. CRC Press, 2005.

[45] Apache Tomcat. *https://tomcat.apache.org/*.

[46] Zhiqiang Xu, Pengcheng Fang, Changlin Liu, Xusheng Xiao, Yu Wen, and Dan Meng. Depcomm: Graph summarization on system audit logs for attack investigation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2022.

[47] R. Yang, S. Ma, H. Xu, Xinagyu. Zhang, and Y. Chen. Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications. *NDSS*, 2020.

[48] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E. Urias, Han Wei Lin, Gabriela F. Ciocarlie, Vinod Yegneswaran, and Ashish Gehani. Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation. *NDSS*, 2021.

[49] Hao Zhang, Danfeng Daphne Yao, and Naren Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014.

[50] Hao Zhang, Danfeng Daphne Yao, Naren Ramakrishnan, and Zhibin Zhang. Causality reasoning about network events for detecting stealthy malware activities. *computers & security*, 2016.

[51] Ningning Zhu and Tzi-cker Chiueh. Design, implementation, and evaluation of repairable file service. *DSN*, 2003.

[52] Zipkin. *https://zipkin.io/*.

APPENDIX A
DISCUSSION ON MAKING AUDITD TRACK TID

Auditd in Linux does not record TIDs by default. To solve this problem, on Linux OS, we can slightly modify the behavior of auditd and make it log TIDs by inserting modules into the kernel. This method is widely used by many works, such as OmegaLog [15].

In detail, we can make auditd record TIDs as follows: observing the *audit_log_exit* function in *linux/kernel/auditsc.c*, we can find that when a syscall event occurs, it will call the function *audit_log_format* in *linux/kernel/audit.c* and log the syscall number. Therefore, we can hook the function *audit_log_format* to call *task_pid_vnr(current)* to get the Linux Kernel PID (TID) under this condition and log it. The core code is shown as follows.

```
/* Function pointer declarations for
the real audit_log_format */
static asmlinkage void (*orig_audit_log_format)
(struct audit_buffer *ab, const char *fmt, ...);

static asmlinkage void hook_audit_log_format
(struct audit_buffer *ab, const char *fmt, ...) {
  va_list args;
  va_start(args, fmt);
  if (strcmp(fmt, "arch=%x syscall=%d") == 0) {
    orig_audit_log_format(
      ab, "tid=%d arch=%x syscall=%d",
      task_pid_vnr(current), args
    );
  } else {
    orig_audit_log_format(ab, fmt, args);
  }
  va_end(args);
}

static struct ftrace_hook hooks[] = {
  HOOK("audit_log_format", hook_audit_log_format,
       &orig_audit_log_format)
};
```

APPENDIX B
BASIC INFORMATION FOR APP 1-7

1) *App 1.* App 1 is not a real application, but a `FastAPI` application specially designed by us to test performance and investigation effectiveness. Its UI elements and backend APIs are designed to be one-to-one. It only contains 5 backend APIs, which correspond to 5 different operations that trigger syscalls, including reading, modifying and deleting operations on files. When each backend request is processed, it will asynchronously sleep for a random span from 0 to 5 seconds to simulate the different processing time of different backend requests. We used Gunicorn to deploy App 1 with 4 processes and timeout is set to 2 minutes.
2) *App 2.* App 2 is a real `FastAPI` project, including 13 backend APIs. We use Gunicorn to deploy App 2, adopt the mode of 4 processes, and set the timeout to 2 minutes.
3) *App 3.* App 3 is a real `Express.js` [7] project, including 42 backend APIs. The project is deployed in a single process (inherently with coroutine).
4) *App 4.* App 4 uses the example application sqla (SQLAlchemy [42] model backend integration examples) provided by the open-source project flask-admin[8], including 24 backend APIs. We use Gunicorn to deploy App 4 and set the time limit to 2 minutes.
5) *App 5.* App 5 uses the open-source project awesome-python3-webapp[9], which is a blog application based on `Aiohttp`, including 11 backend APIs. The project is deployed in a single process (inherently with coroutine).
6) *App 6.* App 6 is a real `Springboot` web application deployed in `Tomcat` server (inherently multi-threaded) with 64 backend APIs. It is deployed in Windows and we used `Procmon` to trace its behavior.
7) *App 7.* App 7 is a real `Express.js` web application with 18 backend APIs. The project is deployed in a single process (inherently with coroutine).

[8] *https://github.com/flask-admin/flask-admin*
[9] *https://github.com/michaellio/awesome-python3-webapp*

APPENDIX C
BASIC INFORMATION FOR APP 8-12

1) *App 8.* App 8 contains CVE-2021-44228[10], the well-known Log4Shell, a vulnerability in `Log4j` that exploited the weakness **Deserialization of Untrusted Data**. It is deployed in `Tomcat` and we exploited the vulnerability to execute 'calc'. TESEC successfully found the attack entrance and the provenance graph is shown in Figure 10 (a).
2) *App 9.* App 9 contains CVE-2021-21315[11], a **Command Injection Vulnerability** in `npm` package `systeminformation`. It is an `Express.js` application and we exploited the vulnerability to upload ssh public key. TESEC successfully found the attack entrance and the provenance graph is shown in Figure 10 (b).
3) *App 10.* App 10 is the python version of XXE-LAB[12]. It is a `Flask` application that contains a **XML eXternal Entity Vulnerability**. We exploited it to read a secret file. TESEC successfully found the attack entrance and the provenance graph is shown in Figure 10 (c).
4) *App 11.* App 11 contains CVE-2016-4437[13], a vulnerability in `Apache Shiro` that exploited the weakness **Improper Access Control**. It is deployed in `Tomcat` and we exploited the vulnerability to execute 'calc'. TESEC successfully found the attack entrance and the provenance graph is shown in Figure 10 (d).
5) *App 12.* App 12 contains CVE-2017-8291[14], a vulnerability in `Artifex Ghostscript` that exploited the weakness **Incorrect Type Conversion or Cast**. It is a `Flask` application and we exploited the vulnerability to read a secret file. TESEC successfully found the attack entrance and the provenance graph is shown in Figure 10 (d).

APPENDIX D
PROVENANCE MODEL

As shown in Table XI and XII, the provenance graph of TESEC consists of 4 types of nodes (representing files, processes/threads, network requests, and UI elements) and edges among them. File and process/thread nodes are the same as those in traditional provenance graphs. The edges between process/thread and file have the same meanings as those in previous works [23].

Network request node, which describes the full content of a network request, is new with our provenance model. The edge between a process/thread and a network request node means that the network request is handled by the corresponding process or thread.

UI element nodes are newly introduced. Each UI element corresponds to a user behavior on UI, and corresponds to a function provided by the testing tool as well as the information of its caller and parameters. The edge from a network request node to a UI element node represents the action that the UI element sends the corresponding network request.

TABLE XI
NODE TYPES IN PROVENANCE MODEL

Node Type	Description
File	same as previous works
Process/Thread	same as previous works
Network Request	full content of a network request
UI Element	interactions with elements on web pages

TABLE XII
EDGE TYPES IN PROVENANCE MODEL

Edge Type	Description
between File and Process/Thread	same as previous works
Process/Thread to Network request	network request is handled by process/thread
Network request to UI Element	UI element sends the network request

[10] *https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-44228*
[11] *https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-21315*
[12] *https://github.com/c0ny1/xxe-lab*
[13] *https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-4437*
[14] *https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2017-8291*

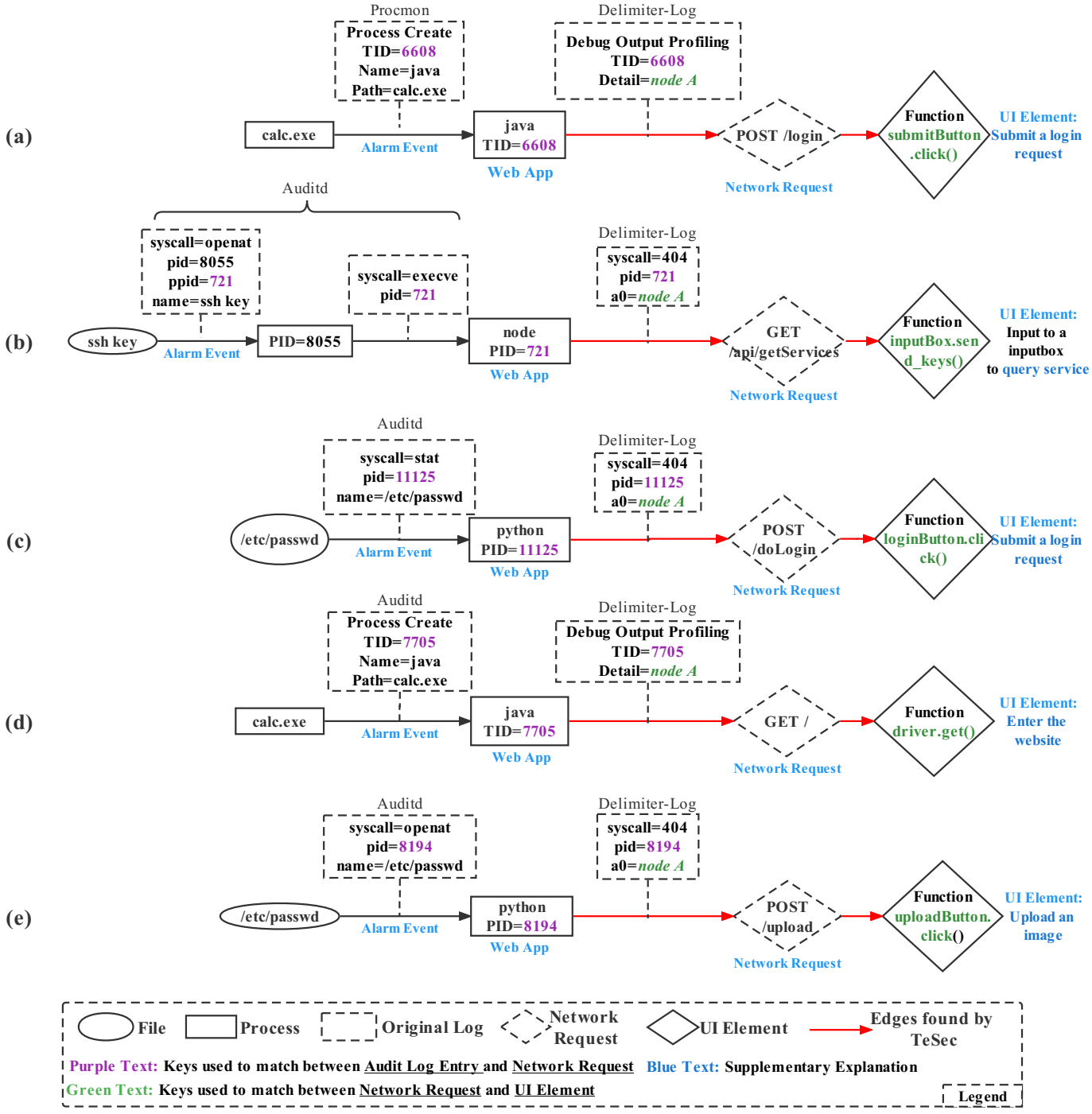

Fig. 10. Provenance Graph for App 8-12.