Detecting and Analyzing Insecure Component Usage*

Taeho Kwon Zhendong Su
University of California, Davis
{kwon,su}@cs.ucdavis.edu

ABSTRACT

Software is commonly built from reusable components that provide desired functionalities. Although component reuse significantly improves software productivity, *insecure component usage* can lead to security vulnerabilities in client applications. For example, we noticed that widely-used IE-based browsers, such as IE Tab, do not enable important security features that IE enables by default, even though they all use the same browser components. This insecure usage renders these IE-based browsers vulnerable to the attacks blocked by IE. To our knowledge, this important security aspect of component reuse has largely been unexplored.

This paper presents the first practical framework for *detecting and analyzing vulnerabilities of insecure component usage*. Its goal is to enforce and support secure component reuse. Our core approach is based on differential testing and works as follows. Suppose that component C maintains a security policy configuration to block certain malicious behavior. If two clients of component C, say a reference and a test subject, handle the malicious behavior inconsistently, the test subject uses C insecurely. In particular, we model component usage related to a policy based on 1) accesses to the configuration state inside the component and 2) the conditional jumps affected by the data read from the state. We utilize this model to detect inconsistent policy evaluations, which can lead to insecure component usage. We have implemented our technique for Windows applications and used it to detect and analyze insecure usage of popular software components. Our evaluation results show that 1) insecure component usage is a general concern and frequently occurs in widely-used software, and 2) our detection framework is practical and effective at detecting and analyzing insecure component usage. In particular, it detected several serious, new vulnerabilities and helped perform detailed analysis of insecure component usage. We have reported these to the affected software vendors, some of whom have already acknowledged our findings and are actively addressing them.

*This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, and the US Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

SIGSOFT/FSE'12 November 10 – 18 2012, Cary, NC, USA.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Reliability, Security

Keywords

Insecure Component Usage, Differential Testing, Testing and Analysis of Real-world Software

1. INTRODUCTION

Component-based development has been a major paradigm for engineering software. In particular, a client application can perform desired functionalities by invoking interface calls of a component. This paradigm allows better code reuse and makes software development more productive. For example, Trident [47], a browser layout engine developed by Microsoft, has been used in IE and many other Windows applications.

Although component reuse has significant benefits, it may lead to security vulnerabilities if a component is not used properly in its client software. The following example, which we first discovered through a manual examination, inspired this research. IE 9 enables an XSS filter by default [23]. However, IE-based browsers, such as IE Tab, use the same browser components as IE, but do not enable the XSS filter. This insecure component usage makes these IE-based browsers vulnerable to XSS attacks. As this example shows, insecure component usage can cause serious vulnerabilities in component-based software. However, this problem has not been much explored. Previous work on component security has focused on designing and developing frameworks for secure component usage [5, 18, 19, 34, 45, 50], detection of insecure components [3, 13, 20, 41], and surveys on component security issues [12, 16].

In this paper, we present a *differential analysis framework* [38] to detect and analyze insecure component usage in component-based software. Here is the key idea behind our framework. Suppose that two applications, a reference `A` (which we assume to be correct and secure w.r.t. component usage) and a test subject `B`, reuse components that check security policies to block malicious activities. If `A` and `B` configure or evaluate the policies inconsistently, `B` may have unprotected runtime execution. In the XSS filter example earlier, IE acts as the reference, and IE Tab uses `URLMON.dll` insecurely because it neither configures the built-in security policies for XSS filter nor utilizes them to block XSS attacks.

To realize our framework, there are two main technical challenges: 1) how to extract the configurations of security policies maintained by a component, and 2) how to detect potential insecure component usage of a client software.

Extracting Policy Configurations. We monitor the writes to component memory space that potentially stores security policy configu-

rations. These memory writes provide several pieces of important information: 1) instructions that configure relevant security policies, 2) locations of the buffers to store the policies, and 3) concrete configuration data. For example, `URLMON.dll` maintains a memory buffer in its global data region to store the URL action policies. IE configures the policies via memory writes.

To check a security policy, an application retrieves its configuration data from the relevant memory buffer and uses the data for comparison. In this case, if a reference and a test subject configure the same security policy in an inconsistent manner, the comparison results are different, making them take different execution paths. Based on this idea, we define missing and incorrect configurations that can lead to insecure component usage.

A *missing configuration* corresponds to the case where the reference only configures and checks a particular set of security policies. Thus, the test subject is vulnerable to attacks that can be blocked by these policies. The earlier XSS filter example belongs to this category. An *incorrect configuration* corresponds to the case where both the reference and the test subject configure and check a particular set of security policies but their different configuration data cause inconsistent subsequent execution paths. For example, while IE enables `FEATURE_HTTP_USERNAME_PASSWORD_DISABLE`, IE Tab does not. The configuration of this security policy is checked by both IE and IE Tab at runtime, but the inconsistent configuration data lead them to behave differently. Specifically, IE Tab allows user names and passwords in a URL address, leading to potential attack vectors for phishing [29]. We provide a detailed analysis of this issue in Section 2.

Detecting Inconsistent Policy Configurations. As we discussed earlier, the inconsistent configuration of a security policy leads to inconsistent subsequent execution patterns. For detection, we capture control flows triggered by the configuration data from the reference and the test subject at runtime and compare them. To capture control flow information, we determine conditional branches whose evaluations are potentially affected by the configuration data via static binary analysis and capture information regarding whether or not each conditional branch has been taken at runtime.

From these observations, we design our differential analysis framework as a three-phase analysis: (**P1**) *detecting potential policy evaluation*, (**P2**) *extracting policy-related execution*, and (**P3**) *detecting inconsistent policy configurations*.

P1: ***Detecting Potential Policy Evaluation.*** This phase detects information related to policy evaluation from dynamic execution of the reference and static properties of a target component. To this end, we detect instructions that read data from component memory space at runtime. Afterward, we perform static forward data slicing to detect conditional jumps that can be affected by the data. If such conditional jumps exist, the data can control the subsequent execution paths at runtime. Thus, the detected instructions potentially read the configuration data and evaluate relevant security policies. We use this information to perform subsequent analyses scalably.

P2: ***Extracting Security Policy-related Execution.*** This phase extracts software execution related to the policy configuration and evaluation performed by the reference and the test subject. To capture the policy configuration, we detect memory writes to component memory space at runtime. Regarding policy evaluation, we log the memory reads and the comparison results on the conditional jumps detected in the previous phase.

P3: ***Detecting Inconsistent Policy Configurations.*** This phase analyzes inconsistent policy-controlled executions between the reference and the test subject to detect missing and incorrect configurations. In particular, we determine whether the conditional jumps relevant to a particular security policy are evaluated consistently.

For evaluation, we implemented our framework for Windows applications and applied it to detect inconsistent policy configurations in reusing popular software components. Our results show that inconsistent policy configurations happen frequently and lead to security vulnerabilities. In particular, we detected several insecure usages of the browser components that disable default protection mechanisms of IE 9. Our framework can also precisely locate root causes of the detected insecure usages, which can help developers fix any detected vulnerabilities and securely reuse software components. The results also show that our framework is scalable. It took less than 15 minutes total to detect inconsistent policy configurations in reusing `URLMON.dll` across all six analyzed browsers.

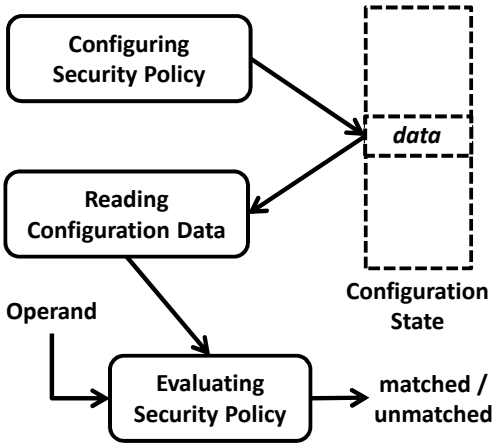


Figure 1: Security policy configuration and evaluation.

This paper makes the following main contributions:

- We introduce and formalize insecure component usage in terms of inconsistent policy configurations and evaluations.
- We develop the first practical framework based on differential analysis to detect inconsistent policy configurations. Our framework works directly on software binaries.
- We implement our framework as a practical tool and evaluate its effectiveness by detecting and analyzing insecure usage of widely-used components in real-world software.

2. DETECTION FRAMEWORK

2.1 Background and Definitions

Security Policy-related Execution. A security policy configuration serves as a key part of software protection because it determines whether or not certain malicious behavior is to be blocked (*e.g.*, IE XSS filter). Figure 1 depicts the runtime process of configuring and evaluating security policies: 1) software maintains its global state and updates the global state to configure a security policy; and 2) when evaluating the policy, the software reads data from the global state and checks whether or not the data match a specified operand. Based on the above description, we formally define security policy-related execution.

DEFINITION 2.1 (CONFIGURATION STATE). *A* configuration state*, denoted by* $M = [\langle k_1, v_1\rangle, \ldots, \langle k_n, v_n\rangle]$*, is an associative array whose key and value pair* $\langle k_i, v_i\rangle$ *corresponds to a configured security policy identifier and its configuration data, respectively.*

We define the configuration and evaluation of a security policy in terms of accesses to a configuration state M.

DEFINITION 2.2 (SECURITY POLICY CONFIGURATION). *For a given configuration state M, a security policy configuration function $conf$ updates M based on a new policy configuration $\langle k, v\rangle$ where k and v correspond to a policy identifier and its configuration data, respectively: $conf(M, k, v) = M'$ such that*

$$M'(k') = \begin{cases} v & \text{if } k' = k \\ M(k') & \text{otherwise} \end{cases}$$

DEFINITION 2.3 (SECURITY POLICY EVALUATION). *Given a configuration state M, a security policy evaluation function $eval$ takes k (a policy identifier) and p (a given operand):*

$$eval(M, k, p) = \begin{cases} \textit{matched} & \text{if } M[k] = p \\ \textit{unmatched} & \text{otherwise} \end{cases}$$

We next define security policy-related execution.

DEFINITION 2.4 (SECURITY POLICY-RELATED EXECUTION). *A security policy-related execution for software S under workload w, denoted by $\pi(S, w)$, is a sequence of policy configurations or evaluations $\pi(S, w) = \langle s_1, \ldots, s_m\rangle$ where $s_i = conf(M, k, v)$ (a policy configuration) or $s_i = eval(M, k, p)$ (a policy evaluation).*

Note that in Definition 2.4, the configuration state M changes at runtime, because S dynamically configures new security policies during its execution over the workload w. Also note that $\pi(S, w)$ provides precise information on policy evaluations during S's execution over w. We now define security policy evaluation patterns.

DEFINITION 2.5 (SECURITY POLICY EVALUATION PATTERN). *For a given $\pi(S, w)$, the security policy evaluation pattern, denoted by $epat(\pi(S, w))$, is the sub-sequence $epat(\pi(S, w)) = [\langle k_i, p_i, eval(M, k_i, p_i)\rangle]$ extracted from the policy evaluations from $\pi(S, w)$ where k_i, p_i, and $eval(k_i, p_i)$ correspond to a policy identifier, an operand for its policy evaluation, and the evaluation result (*matched *or* unmatched*) respectively.*

Insecure Component Usage. For a reference R and a test subject T, we define two types of inconsistent policy configurations: *missing* and *incorrect configurations*.

DEFINITION 2.6 (MISSING CONFIGURATION). *The test subject T misses the configuration of a policy k evaluated by the reference R if $\exists p, s(\langle k, p, s\rangle \in epat(\pi(R, w))$ and $\forall p, s(\langle k, p, s\rangle \notin epat(\pi(T, w))$.*

DEFINITION 2.7 (INCORRECT CONFIGURATION). *The test subject T incorrectly configures a policy k if $\exists p, r_1 \neq r_2(\langle k, p, r_1\rangle \in epat(\pi(R, w)) \wedge \langle k, p, r_2\rangle \in epat(\pi(T, w)))$.*

Inconsistent policy configurations can lead to unprotected software execution; we define it as follows.

DEFINITION 2.8 (UNPROTECTED SOFTWARE EXECUTION). *Suppose that a security policy k blocks a malicious behavior Φ at runtime. Given a reference R, a test subject T, and a common workload w, T is unprotected w.r.t. k if R blocks Φ but T does not.*

Suppose that a component C maintains its configuration state M at runtime. If a client using C configures a policy stored by M insecurely, it can be unprotected w.r.t. the policy. We next define insecure component usage.

DEFINITION 2.9 (INSECURE COMPONENT USAGE). *Suppose that a component C maintains a security policy k that blocks a malicious behavior Φ at runtime. For a reference R and a test subject T that use C, T insecurely uses C w.r.t. k if T is unprotected w.r.t. k.*

2.2 Overview

To detect inconsistent policy configurations formalized in Definitions 2.6 and 2.7, it is necessary to analyze how security policies are set and enforced in code. However, it is challenging, because 1) most components are distributed as binaries, and 2) it is difficult to know which memory locations are used for security policy configuration and evaluation. To address this issue, at the high-level, we design our framework as two phases: *runtime extraction* and *offline detection*. In the runtime extraction phase, we instrument executions of a reference and a test subject to capture security policy-related executions (Definition 2.4). We perform an offline analysis to detect insecure component usage in the captured executions.

Although the high-level approach appears straightforward, the main challenge is how to perform scalable and precise analysis. For example, IE performs millions of memory accesses at runtime, and it is practically infeasible to instrument and analyze all of them.

To address this scalability issue, our framework uses the following optimizations: 1) instrumenting target component execution, 2) filtering irrelevant memory accesses, and 3) performing preliminary analysis on policy evaluation.

Instrumenting Target Component Execution. Instrumenting all instructions executed by an application at runtime suffers from poor scalability. To mitigate this issue, our framework only instruments components of interest at runtime, because the configuration state maintained by the components is generally accessed by their code. Suppose that the configuration state M is maintained by a component A. When other components access M, they generally invoke relevant interface calls of A that access M.

It is possible that other components can directly access M, but this cannot be used in component-based software development because the location of M cannot be reliably resolved. For example, the base addresses of the components loaded at runtime often change [42, 52], making locations of their global data regions inconsistent.

Filtering Irrelevant Memory Accesses. As we discussed, capturing all accesses to arbitrary memory space is not feasible. To mitigate this issue, we filter the memory accesses that are unlikely to be relevant to security policy-related executions.

Our *key insight* is that the code executed by any thread can access the configuration state maintained by a component at runtime. Suppose that two different threads configure and evaluate a particular security policy. In this case, both threads should access the same memory location. Otherwise, integrity issues on the configuration data would arise. Based on this observation, we filter the logging of accesses to thread-specific memory space, such as the stack.

Performing Preliminary Analysis on Policy Evaluation. According to Figure 1, policy evaluations are conducted by 1) reading data from the configuration state and 2) determining whether or not the data is matched with a specified operand. In order to detect inconsistent policy configurations, it is necessary to capture the results of policy evaluations at runtime. However, it again suffers from scalability problems, because the policies are generally evaluated by conditional jumps such as `JNE`, which are executed frequently to determine program flow at runtime. Thus, instrumenting all conditional jumps is infeasible in practice.

To address this problem, we perform a preliminary analysis to detect those conditional jumps affected by data reads from the configuration state. Our observation is that the evaluation of the conditional jumps are affected by the data reads from the configuration state (see Figure 1). We detect these conditional jumps via dynamic and static binary analyses. Specifically, we dynamically capture the memory reads from the configuration state under a given workload. Then we extract the conditional jumps potentially affected by them via static binary analysis. We use information on the memory reads

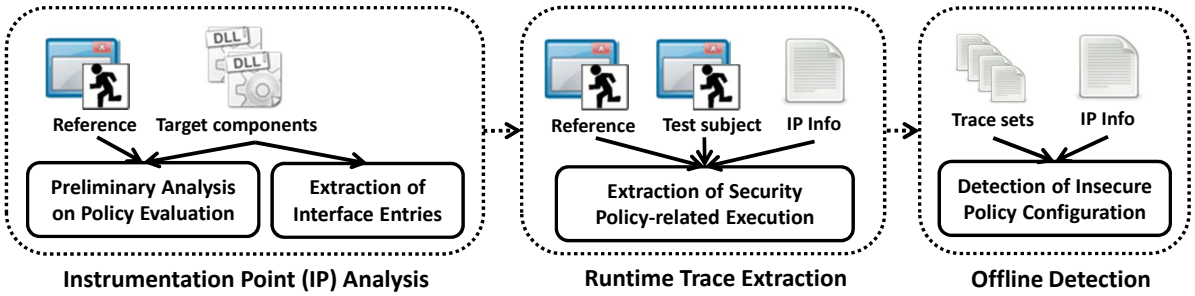


Figure 2: Detection framework.

and the conditional jumps to reduce logging of policy evaluations in the subsequent phase.

Based on these optimizations, we present our framework in Figure 2. The following sections illustrate details of each phase in our framework using a running example. In particular, we detect insecure component usage to allow potentially malicious URL addresses in IE-based browsers. In the example, we use `WININET.dll` as a target component and access `http://www.microsoft.com` as the workload for detection.

2.3 Instrumentation Point Analysis

Preliminary Analysis on Policy Evaluation. As discussed, instrumenting all memory reads and conditional jumps is not feasible. To address this problem, we detect them in advance and instrument their executions in the subsequent analysis. Our key observation is that there exists data flow between the memory reads and the conditional jumps. According to Figure 1, the policy evaluation is affected by the configuration data read. Based on this, we locate the instructions relevant to the policy evaluation via dynamic and static program analyses. In particular, we 1) dynamically instrument the execution of target components to detect reading data from the configuration state and 2) perform static forward data slicing w.r.t. the detected memory reads to locate relevant jumps. Note that we assume heap or global data regions may contain the configuration state, because they are not thread-specific (Section 2.2).

Figure 3(a) shows an example of policy-controlled execution by `WININET.dll` during the workload. In particular, `i1` reads the configuration data `_GlobalProcessDisableUserPswdForHttp` stored in the global data region of `WININET.dll` and determines whether or not the data is equal to zero. The evaluation result affects the invocation of the `GetUrlAddress` function at `i5` by deciding one of its parameters, *i.e.*, the value of `eax`.

To detect the instructions relevant to policy evaluation, we capture the instructions to read data from the global data region in `WININET.dll` during the workload via dynamic binary instrumentation (*i.e.*, `i1`). Then we extract the static forward slice w.r.t. the data read by `i1`. In this case, the slice contains `i1` and `i2`, because the `cmp` instruction at `i1` sets `ZF` according to the comparison result, and `jz` takes `ZF` to determine the next instruction to execute.

Once the slices w.r.t. the detected memory reads are extracted, we analyze them to locate the instructions relevant to the policy evaluation. In particular, suppose that S is a forward data slice w.r.t. a memory read instruction I. In this case, we consider I relevant to the policy evaluation only if S contains conditional jumps. This is because policy evaluations are generally performed by both memory reads and relevant conditional jumps (see Figures 1 and 3(a)). Based on this idea, we determine the memory reads and the conditional jumps in its forward slice as the instrumentation points in the subsequent analysis.

Note that we only perform this preliminary analysis for the reference software. This is because our framework does not detect security policies configured only in the test subject (Definitions 2.6 and 2.7). To detect them, we swap the test subject with the reference and repeat the analysis.

Extracting Interface Entries. As we have discussed, when a component A maintains the configuration state M, other components generally access M by invoking interface calls exported by A. Thus, information on the invoked interface calls provides us with detailed insight on insecure component usage.

To capture the information, our framework dynamically instruments the entry points of the interface calls exported by the target components. To this end, it is necessary to determine the interface entries as the instrumentation points. However, it is difficult to locate them at runtime, because the instructions at the entries are also frequently executed by non-entry code at runtime. For example, while the `push` instruction is often executed at the entry point as part of a function prologue, it is also used for parameter passing. To address this problem, we perform static binary analysis to extract the interface call entries and pass them into the subsequent analysis as the instrumentation points.

Our key observation is that the addresses of the interface call entries are generally stored in data tables that can be read from outside of the component. Because components are developed as position-independent code, they generally support memory chunks that can be accessed by other components to resolve the virtual addresses of desired interface calls at runtime. For example, the PE [39] and ELF [15] formats have Export Table and Procedure Linkage Table to provide dynamic linking, respectively. We statically analyze data reference to the entries of all functions in the target components. We consider the function entries that have such data references as the instrumentation points to capture interface call invocations.

2.4 Runtime Trace Extraction

This phase extracts detailed information on security policy-related executions of the target components during the workload run by both the reference and the test subject.

In particular, we instrument the runtime execution of the target components to record the policy configurations, their evaluations, and invocations of the interface calls to the target components. We store the captured information to files for use in our offline analysis.

Policy Configuration. According to Definition 2.2, an application specifies security policies via memory writes to its configuration state. To capture policy configurations, we instrument the runtime information of the target components that perform data writes to non-thread specific memory regions (Section 2.2). During instrumentation, we log addresses of the memory writes, values of the data written, and addresses of the memory written.

Figure 3(b) shows a policy configuration example by `WININET.dll` with the given workload. In particular, `i1-i6` initialize `eax` by determining whether `FEATURE_HTTP_USERNAME_PASSWORD_DISABLE` is enabled.

```
...
i1   cmp  _GlobalProcessDisableUserPswdForHttp, 0
i2   jz   i6
i3   xor  eax, eax
i4   inc  eax
i5   call GetUrlAddress(*,*,*,*,*,*,*,*,*,eax,*,*)
...
i6   xor  eax, eax
i7   jmp  i5
...
```

(a) Evaluation.

```
...
i1  push offset g_FEATURE_HTTP_USERNAME_PASSWORD_DISABLE
i2  call _CoInternetIsFeatureEnabledInternal
i3  dec  eax
i4  neg  eax
i5  sbb  eax, eax
i6  neg  eax
i7  mov  _GlobalProcessDisableUserPswdForHttp, eax
...
```

(b) Configuration.

Figure 3: Policy-related code example.

Next `i7` writes the value of `eax` to a memory buffer `_GlobalProcessDisableUserPswdForHttp` in the global data region of `WININET.dll`. In this case, our framework instruments the execution of `i7` and logs the following information: address of `i7`, address of `_GlobalProcessDisableUserPswdForHttp`, and value of the `eax`.

Policy Evaluation. To capture information on policy evaluations, our framework only instruments executions of those instructions detected by the preliminary analysis from the previous phase. Note that this allows us to significantly reduce the instrumentation points for capturing policy evaluations (see Section 2.2).

Regarding the information to be captured, consider the following code execution for evaluating a policy: an instruction I reads a data D from a non-thread specific memory region Mem, and D determines whether a conditional jump $Cond$ is taken or falls through. In this case, our framework logs 1) the address of I, 2) the value of D, 3) the address of Mem, 4) the address of $Cond$, and 5) the evaluation result of $Cond$ (*i.e.*, taken or fall-through). For example, when instrumenting Figure 3(a), we capture the address of `i1`, the value of `_GlobalProcessDisableUserPswdForHttp`, the address of the memory read, and the evaluation result of `i1`.

Interface Call Entries. We capture invocations of the interface calls to the target components by instrumenting their statically-detected entry points. However, it is possible for a component to invoke its interface call at runtime. To detect this, we analyze the return addresses of invoked interface calls, which are stored on the top of the stack. In particular, suppose that we instrument an interface call entry f of a component C. In this case, we log f only if the return address of f is not part of the memory space corresponding to C. Based on this approach, we can precisely detect invocations of the interface calls to target components at runtime.

2.5 Offline Detection

From the previous analyses, we obtain the execution traces of the target components by the reference and the test subject under the given workload. Each trace contains a sequence of detailed information for each software with the following runtime information: 1) the policy configurations, 2) the policy evaluations, and 3) the invocations of the interface calls on the target components. Our offline phase detects inconsistent policy configurations from the traces as follows:

Extracting Policy Configurations and Their Evaluations. For each trace, we extract information on the configuration and the evaluation for each security policy. To this end, we first track the memory access patterns for each captured data address. For example, `i7` in Figure 3(b) writes a configuration data to a memory region `_GlobalProcessDisableUserPswdForHttp`, and `i1` in Figure 3(a) reads the data. Based on this memory access pattern, we can infer the instructions for configuring security policies and reading their configuration data.

Once the instructions that read the configuration data are located, we can extract the evaluation results of their relevant conditional jumps. In particular, we sequentially search for the relevant conditional jumps, starting from the instructions, until the next read access of the configuration data is found. Next we retrieve their evaluation results captured during the instrumentation. Note that the result of the preliminary analysis provides us with the conditional jumps relevant to the instructions. For example, because `i1` affects the evaluation of `i2` (see Section 2.3), `i2` can be located by checking the instructions that follow `i1` in the trace.

Detecting Inconsistent Policy Configurations. The previous analysis step provides us with the policy configurations and their evaluation results for the reference and the test subject. We use this information to detect inconsistent policy configurations formalized in Definitions 2.6 and 2.7. Missing configurations are detected by finding those policy configurations and the associated evaluations that are only present in the reference, and incorrect configurations are detected by finding inconsistent evaluations of the relevant conditional jumps. In particular, a security policy is configured incorrectly in the test subject if the following conditions are satisfied: 1) both the reference and the test subject configure the same security policy and read its configuration data; 2) the policy evaluation results on the data are different. For example, while IE Tab takes a jump at `i2` in Figure 3(a), IE just continues the execution. This inconsistency shows that IE Tab does not enable `FEATURE_HTTP_USERNAME_PASSWORD_DISABLE` whose configuration is stored in `WININET.dll`. This incorrect configuration makes IE Tab allow user names and password in URL address, which is blocked by IE [29]. Thus, IE Tab misuses `WININET.dll` w.r.t. the security policy `FEATURE_HTTP_USERNAME_PASSWORD_DISABLE`, making it vulnerable to phishing attacks [29].

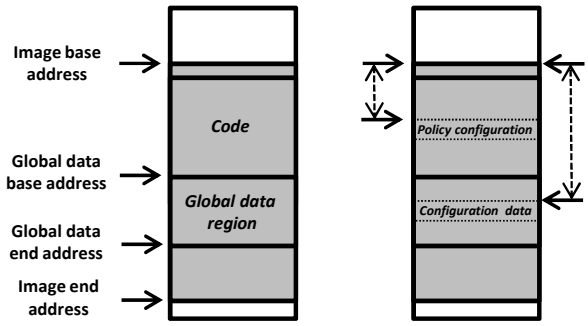
Helping Developers to Securely Use Components. Our framework can extract interface calls for policy configurations and their evaluations. Because we capture all invocations of the interface calls to the target components, we can infer which interface calls perform policy configurations or their evaluations. For example, IE Tab invokes the `InternetSetOptionA` function to execute `i7` in Figure 3(b) for policy configuration.

This interface-level information can help developers to use components securely. For example, IE configures its policy by invoking the `InternetQueryOptionW` function, leading to its correct policy configuration. Although both IE and IE Tab evaluate the policy while invoking `HttpSendRequestW`, the interface calls that configure the policy are different. This information can guide developers of IE Tab to securely use `WININET.dll`.

3. IMPLEMENTATION

We have implemented our technique for Microsoft Windows applications. This section presents details on our implementation.

Overview. According to Figure 2, our framework consists of three



(a) Base and end addresses. (b) Relative offset.

Figure 4: Memory address space of loaded target component.

phases: 1) instrumentation point analysis, 2) runtime trace extraction, and 3) offline detection. The main components for the first two phases are dynamic binary instrumentation and static binary analysis. To this end, we use Pin [36] for runtime instrumentation and have developed plugins for IDA Pro [21] (a state-of-the-art commercial binary disassembler) by using IDAPython [22] for binary analysis. When instrumenting each policy-related execution, we record not only the information of the captured instructions but also process and thread identifiers at runtime. This makes our offline analysis independent of thread interleavings. For the offline phase, we have developed Python scripts to analyze the traces obtained from the earlier phases.

Runtime Trace. As we discussed in Section 2.2, an application performs many memory accesses and conditional jumps. Although we filter those irrelevant ones, the number of captured instructions is still large. For example, when IE accesses the Google web page, we dynamically captured a large number (332,756) of memory accesses and conditional jumps. To store this large amount of information efficiently, we dump the captured information as binary data. As an example, we designed a data structure to store conditional jumps that contains the following information: an identifier for the conditional jump, process and thread identifiers, the address of the conditional jump, and its evaluation result. When capturing this information on a conditional jump, we fill the data structure and stores it as binary data of twenty bytes. Using this optimization, we can effectively analyze the large captured information in practice.

Instrumenting Component Code Execution. Our framework only instruments executions of target components at runtime. To this end, we dynamically instrument the loading of each image and determine whether or not the image is one of the target components. If so, we use the base and end addresses of the image (see Figure 4(a)) to determine the instructions to instrument. In particular, we instrument executions of those instructions whose virtual addresses are part of the memory spaces of the target components. Note that a target component is loaded before its instructions are executed.

Filtering Irrelevant Memory Accesses. To capture policy configurations and evaluations, we focus on memory accesses to the global data region of a target component. To this end, when the target component is loaded, we extract the base and end addresses of its global data region (see Figure 4(a)). We use this information to detect those instructions that access the global data region.

Logging Virtual Address Information. We capture information on virtual addresses for use by the offline detection. For example, we log information on those virtual addresses for reading or writing configuration data. However, we cannot use virtual addresses alone for offline detection because the same virtual address may not refer to the same location in the target component. For example, suppose that a component C is loaded by the reference and the test subject at the memory spaces starting with different base addresses [42, 52]. In this case, the same virtual address in the offline phase does not refer to the same instructions or memory buffers.

To address this issue, when capturing a virtual address at runtime, we compute its relative offset from the base address of the loaded target component (*e.g.*, Figure 4(b)) and log it. Using this approach, we can precisely extract the virtual address information independent of the base address of the loaded target component.

4. EMPIRICAL EVALUATION

In this section, we evaluate how effective our technique is for detecting and analyzing insecure component usage in popular Windows applications. We show that 1) our framework can automatically detect inconsistent policy configurations in real-world software, and inconsistent policy configurations are prevalent and constitute a general security and reliability issue (Section 4.1); 2) our in-depth study of selected inconsistencies reveal new, serious vulnerabilities in widely-used software (Section 4.2); and 3) our framework can be effectively used for root-cause analyses to understand the detected inconsistent policy configurations and vulnerabilities (Section 4.3).

4.1 Prevalence of Inconsistent Configurations

to detect inconsistent policy configurations in real-world software. In particular, we have analyzed applications using widely-used components (such as the IE browser components and the Flash Player) and evaluated how our chosen reference programs and test subjects differ in terms of policy configurations under various workloads. Table 1 gives the detailed information on the analysis of the IE browser components, in particular how many detected inconsistent policy configurations in the Trident-based browsers w.r.t. their respective reference program (*i.e.*, IE) and workloads. [1] Our results show that inconsistent policy configurations frequently occur in real-world, widely used applications. Note that all the reported inconsistencies are *real* and *detected fully automatically* by our tool by capturing and comparing inconsistent security policy evaluation patterns.

According to Definition 2.9, inconsistent policy configurations can lead to insecure component usage. Our framework automatically detects inconsistent policy configurations. A detailed analysis is needed to understand how security relevant they are. We perform such a detailed analysis of insecure component usages of major IE components [47] (*i.e.*, `MSHTML.dll`, `URLMON.dll`, and `WININET.dll`) in real-world IE-based browsers. We consider IE as a reference and the following browsers as test subjects: IE Tab 2 [24], Lunascape 6 [37], Slim Browser 5.01 [43], Green Browser 5.8 [17], WebbIE 3.14 [51], and Enigma Browser [14]. The tested Trident-based browsers are in wide use. For example, IE Tab is among the most popular plugins for both Firefox and Chrome, and has millions of downloads and users [25, 26, 46], and Lunascape has more than 20 million downloads and millions of users.

We next describe the new security vulnerabilities we discovered. We have reported these problems to the affected software vendors, and some of them such as Lunascape and IE Tab have already acknowledged our findings. Since we were able to manually trigger all these reported vulnerabilities, they constitute real, and some of which very serious, security concerns. We also provide further discussions in Section 4.5.

4.2 New Vulnerabilities Discovered

As discussed in Section 2, we can utilize our framework to detect the security vulnerability where the test subjects allow an insecure

[1] Due to space constraints, we include the result of our empirical evaluation on other components such as Flash Player and Quick Time Player in the tech report [35].

Reference	Workload	Component	Test subject	Inconsistent policy configuration		Total
				Missing	Incorrect	
Internet Explorer 9	connect to microsoft.com	URLMON.dll	IE Tab 2	176	28	204
			Lunascape 6	167	33	200
			Slim 5.01	197	36	233
			Green 5.8	188	26	214
			WebbIE 3.14	175	27	202
			Enigma	190	25	215
		WININET.dll	IE Tab 2	215	18	233
			Lunascape 6	272	21	293
			Slim 5.01	229	19	248
			Green 5.8	235	14	249
			WebbIE 3.14	217	11	228
			Enigma	187	20	207
	login to gmail.com	URLMON.dll	IE Tab 2	43	12	55
			Lunascape 6	81	10	91
			Slim 5.01	45	16	61
			Green 5.8	83	12	95
			WebbIE 3.14	112	13	125
			Enigma	33	17	50
		WININET.dll	IE Tab 2	151	4	155
			Lunascape 6	102	5	107
			Slim 5.01	148	3	151
			Green 5.8	138	6	144
			WebbIE 3.14	153	3	156
			Enigma	128	3	131
		MSHTML.dll	IE Tab 2	16	1	17
			Lunascape 6	16	1	17
			Slim 5.01	14	1	15
			Green 5.8	15	1	16
			WebbIE 3.14	16	1	17
			Enigma	16	1	17

Table 1: Inconsistent policy configurations in test subjects reusing IE components w.r.t. given workloads.

URL scheme that can be exploited by phishing attacks [29]. This section illustrates the effectiveness of our framework by detecting additional security vulnerabilities. Our high-level approach is as follows. We first capture inconsistent policy configurations from a given workload. Next we analyze them for detecting potential insecure component usage and then manually trigger them for validation. For evaluation, we consider the accesses to the URLs for Microsoft homepage and gmail account as workloads. Table 1 shows the inconsistent policy configurations our tool detected in the test subjects under the given workloads. Next we describe the security problems caused by them and our analysis approach.

Insecure Configuration of URL Security Zone. As a protection mechanism, IE categorizes URL namespaces into five types of URL security zones (*i.e.*, Local Intranet, Trusted Sites, Internet, Restricted Sites, and Local Machine). Each zone has a different trust level [1] to determine whether or not a URL action is allowed. For example, while Internet zone allows the execution of script code, Restricted Site zone does not.

This privilege-based protection mechanism can cause security vulnerabilities. Suppose that a web page on a particular zone accesses resources on less restrictive zones. In this case, when accessing the web page, privilege escalation happens, called *Zone Elevation*. This security vulnerability has been exploited by real-life attacks[2] based on Cross Zone Scripting [11]. To mitigate this issue, IE blocks the Zone Elevation [31]. However, the test subjects do not block it and are vulnerable to these attacks. Next we describe how to use our framework to detect these vulnerabilities in the test subjects.

[2]Examples include MS05-001, MS05-014, MS08-048, and CVE-2008-2281.

Under the gmail workload, `MSHTML.dll` stores the configuration of the security policy on Zone Elevation and checks it at runtime. Figure 5(a) illustrates its detailed code and operates as follows. First, `i1`–`i3` invoke an interface call `CoInternetIsFeatureEnabled` to `URLMON.dll`, which determines whether or not the current process enables `FEATURE_ZONE_ELEVATION` [32]. Then `i4`–`i5` initialize the memory buffer `byte_6402C6C4` based on the result of the function. The stored value is evaluated to check the configuration of the feature in `i6`–`i7`. In particular, if the feature is enabled, the conditional jump in `i7` falls through in the execution. Otherwise, it takes the jump.

For IE and the test subjects, we analyzed inconsistencies in `i7` to detect incorrect configurations. We found that only IE enables this feature and the test subjects allow Zone Elevation. To validate this finding, we developed a trusted web site having an `<iframe>` tag to a local HTML file. When accessing the site, we observed that the zone elevation is indeed only successful for the test subjects.

In order to launch the cross zone scripting attacks, it is necessary to run script code in a local HTML file. To block this malicious behavior, IE adopts a default protection mechanism, called Local Machine Zone Lockdown (LMZL) [30], which configures more restrictive security policies on particular URL actions. For example, IE disallows the execution of any script code in local HTML files by default. During our validation, we identified that the test subjects do not adopt LMZL, allowing the execution of local script code.

Thus, this insecure component usage can lead to serious vulnerabilities. In particular, the disabled `FEATURE_ZONE_ELEVATION` and the missing LMZL make the test subjects vulnerable to cross zone scripting attacks.

```
...
i1  push    2 // GET_FEATURE_FROM_PROCESS
i2  push    1 // FEATURE_ZONE_ELEVATION
i3  call    CoInternetIsFeatureEnabled
i4  cmp     eax, 1
i5  setnz   byte_6402C6C4
...
i6  cmp     byte_6402C6C4, 0
i7  jz      short loc_6397DAB1
...
```

(a) Zone elevation.

```
...
i1  mov  edi, offset g_FEATURE_ALLOW_LONG_INTERNATIONAL_FILENAMES
i2  push edi
i3  call CoInternetIsFeatureEnabledInternal
i4  neg  eax
i5  sbb  eax, eax
i6  inc  eax
i7  mov  _GlobalAllowLongIntlFileNames, eax
...
i8  cmp  _GlobalAllowLongIntlFileNames, 0
i9  jz   i10
...
```

(b) Long filename handling.

Figure 5: Policy configuration and evaluation.

Incorrect Handling of Long File Name. When IE 9 beta and the test subjects connect to `microsoft.com`, they configure & evaluate a policy on `FEATURE_ALLOW_LONG_INTERNATIONAL_FILENAMES` at runtime. Figure 5(b) illustrates the relevant code of `WININET.dll` for this policy configuration and evaluation. In particular, `i1`–`i6` determine whether or not the feature is enabled by invoking `CoInternetIsFeatureEnabledInternal`. Later `i7` writes the result to the memory buffer `_GlobalAllowIntlFileNames` in the global data region of `WININET.dll`. For evaluation, `i8` reads the configuration data from the `_GlobalAllowIntlFileNames`, and `i9` evaluates the configuration by comparing its value of the data with zero. According to our analysis, the test subjects take the branch at `i9`, but IE 9 beta falls through. This shows that only IE 9 beta enables the feature.

This feature is related to the maximum path length limitation [40]. In particular, for a given file, its fullpath length cannot be longer than 256. Suppose that IE downloads and opens a file whose name having non-ASCII characters. In this case, the previous IE releases store the file to the temporary folder by encoding its name based on UTF-8 [49] and opens it based on the encoded fullpath. However, the length of its fullpath is often longer than the given limit. For example, when a `.xlsx` file whose name is composed of 17 ASCII and 12 Korean characters is downloaded, the length of its encoded fullpath is larger than 256. In this case, Microsoft Excel 2010 cannot open the downloaded file [53]. To mitigate this issue, Microsoft released a hotfix KB982381, and recent IE releases changed the encoding scheme for international file names. We confirmed the detected inconsistent configuration because the test subjects cannot download and open the `.xlsx` file. Thus, incorrect policy configurations may also cause compatibility issues.

4.3 Root-Cause Analysis of Vulnerabilities

As we discussed in Section 4.2, the insecure configuration of the URL zones leads to security vulnerabilities. To mitigate this issue, it is necessary to configure and evaluate the URL action policies in a secure manner. However, we observe that third-party developers often insecurely reuse the IE browser components without considering this issue, which makes the test subjects disable these protection mechanisms of IE. For example, although IE supports XSS and Phishing filters [23, 44] by default, the test subjects neither configure relevant security policies nor block the malicious behavior. To address this problem, it is necessary to understand the root cause of this insecurity. In this section, we present how to use of our framework to analyze insecure URL action policies.

Evaluation of URL Action Policies. According to MSDN, the evaluation of URL action policies is performed by certain interface calls exported by `URLMON.dll`. For example, `ProcessUrlAction` [27] determines whether or not a specified action for a particular URL is allowed. Based on this information, we reverse engineered such interface calls to analyze the detailed process of evaluating URL action policies.

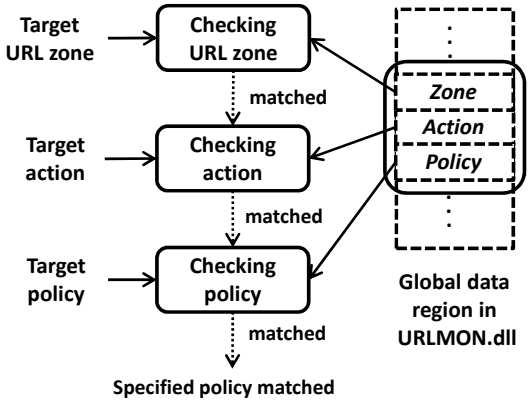


Figure 6: Evaluation of URL action policies.

Figure 6 shows the high-level overview of this evaluation. In particular, `URLMON.dll` maintains URL action policies as a list of memory buffers in its global data region. Each buffer contains information on a URL zone, a URL action (*e.g.*, downloading signed ActiveX), and its policy (*e.g.*, allow). The evaluation of the URL action policies is performed by an internal function, which is invoked by several interface calls at runtime. In particular, the function takes three parameters (*i.e.*, a URL zone, a URL action, and a policy to check) and iterates through the memory buffers to locate the configuration whose data are matched with the parameters. If such a memory buffer is found, the function returns true, showing that the specified URL action policies are matched with the current configuration setting. To extract the URL action polices checked by the reference and the test subjects at runtime, we can use our framework to infer the detailed information on policy evaluation. This is because 1) the configuration data is stored in the global memory space, and 2) matching the parameters with the configuration data is performed by conditional jumps affected by the data read.

Based on the above observations, we analyze the runtime traces obtained in the second phase of Figure 2 to extract the evaluations performed by the reference and the test subjects. To this end, we first locate the matched conditional jumps on the target policy. Then we traverse the traces backward to find the consecutive conditional jumps that check the target action and URL zone. According to Figure 6, the data reads for evaluating the detected conditional jumps correspond to the target URL zone, action, and its policy. Using this approach, we can extract the evaluations of URL action policies performed by the reference and the test subjects. In the following

URL action	XSS filter (Internet)		Phishing filter (Internet)		Script execution (Local machine)	
	Reference	Test subjects	Reference	Test subjects	Reference	Test subjects
URLACTION_DOWNLOAD_SIGNED_ACTIVEX	✓	✗	✓	✗	✗	✗
URLACTION_DOWNLOAD_UNSIGNED_ACTIVEX	✓	✗	✓	✗	✗	✗
URLACTION_ACTIVEX_OVERRIDE_OBJECT_SAFETY	✓	✓	✓	✗	✗	✗
URLACTION_SCRIPT_RUN	✓	✓	✓	✓	✓	✗
URLACTION_SCRIPT_XSSFILTER	✓	✗	✗	✗	✗	✗
URLACTION_HTML_INCLUDE_FILE_PATH	✓	✗	✗	✗	✗	✗
URLACTION_SHELL_VERB	✓	✗	✓	✗	✗	✗
URLACTION_SHELL_EXECUTE_HIGHRISK	✓	✗	✓	✗	✗	✗
URLACTION_COOKIES_ENABLED	✓	✓	✗	✗	✗	✗
URLACTION_BEHAVIOR_RUN	✓	✓	✓	✗	✗	✗
URLACTION_FEATURE_MIME_SNIFFING	✓	✗	✓	✗	✗	✗
URLACTION_FEATURE_DATA_BINDING	✓	✓	✓	✓	✓	✗
URLACTION_ALLOW_APEVALUATION	✗	✗	✓	✗	✓	✗
URLACTION_INPRIVATE_BLOCKING	✓	✓	✓	✓	✓	✗
URLACTION_ALLOW_STRUCTURED_STORAGE_SNIFFING	✗	✗	✓	✓	✗	✗

Table 2: Evaluated URL action policies for XSS and Phishing filters / local script execution.

Workload	Target component	IP analysis (s)	Runtime trace extraction (s)							Offline phase (s)
			IE 9	IE Tab 2	Lunascape 6	Slim 5.0.1	Green 5.8	WebbIE 3.14	Enigma	
microsoft.com	URLMON.dll	73.8	169.7	264.6	792.0	865.4	354.7	444.8	288.5	152.2
	WININET.dll	108.2	265.6	483.9	290.1	165.2	247.0	300.2	267.2	237.7
gmail account	URLMON.dll	201.1	249.0	197.7	221.4	159.9	258.4	238.8	188.8	518.5
	WININET.dll	270.5	583.1	348.0	327.0	315.8	282.8	384.0	349.2	536.5
	MSHTML.dll	1,852.2	1,902.2	1,909.2	1,774.7	1,698.6	1,462.1	1,774.2	1,436.0	59.6
XSS filter	URLMON.dll	82.3	79.8	111.6	92.5	142.9	78.4	74.7	117.2	106.0
Phishing filter	URLMON.dll	76.5	104.6	179.6	89.3	75.8	71.4	71.3	79.8	98.6
Local script run	URLMON.dll	190.9	105.5	67.1	86.9	83.7	67.7	79.6	66.5	89.3

Table 3: Execution time for each analysis phase.

sections, we discuss and analyze the security vulnerabilities caused by insecure configurations of URL action policies.

Disabled XSS and Phishing Filters. Recent IE releases have supported the XSS filter [23] and the Phishing (or SmartScreen) filter [44] by default. These mechanisms can effectively protect users from unknown XSS and phishing attacks. We confirmed that the test subjects do not enable these protection mechanisms even though they use the same browser components. To analyze these security vulnerabilities, we access malicious web sites that trigger these filters (*i.e.*, our workloads). Then we apply our framework to capture the runtime traces of the reference and the test subjects running under the given set of workloads. We next extract the evaluations of these URL action policies using our approach discussed earlier in the section. Table 2 shows the evaluated URL action policies for the Internet zone under our workloads, where ✓ represents the case where the corresponding policy has been evaluated, and ✗ represents that the corresponding policy has not been evaluated. It is interesting that the URL action policies relevant to XSS and Phishing filters are evaluated only when these filters are enabled. This information helps pinpoint the code relevant for evaluating the policies.

In the case of XSS filter, `MSHTML.dll` calls an internal function `IsXssFilterEnabled`, which invokes an external function `ProcessUrlAction` exported by `URLMON.dll`, to check whether the XSS filter is enabled. As for the Phishing filter, `MSHTML.dll` calls an internal function `CMarkup::ProcessUrlAction2` to invoke an interface call `ProcessUrlActionEx2` to `URLMON.dll`, which checks whether or not the Phishing filter is enabled. The information on the caller-callee relationship can be a starting point for analyzing software behavior relevant to these URL actions. Note that the top of the stack at the interface entries contains the address to be returned after invoking the interface call.

Disabled Local Machine Zone Lockdown. As we have discussed in Section 4.2, the test subjects allow local script execution, making them vulnerable to cross zone scripting attacks. To analyze this vulnerability, we run local script code using an external script file as workload and repeat the analysis steps earlier for the XSS and Phishing filters. Table 2 shows the evaluated URL action policies for Local Machine during the workload. According to our result, all test subjects execute the local script code without evaluating any security policy on its action. However, IE evaluates the security policies on the potentially malicious behavior and blocks it. For example, IE blocks the execution of local script code, protecting IE from the cross zone scripting attacks. Similar to the Phishing filter case, the function `CMarkup::ProcessUrlAction2` of `MSHTML.dll` invokes the interface call `ProcessUrlActionEx2` to evaluate the security policy on `URLACTION_SCRIPT_RUN` at runtime.

4.4 Performance

Table 3 shows the execution time for each phase of our analysis (Section 4.2). All experiments were done on a Core2 Duo 2.40GHz processor with 4GB RAM. The results show that our framework can easily scale to real-world applications. For example, it can detect, in about 2 hours, inconsistent configurations of the three components by the 6 real-world IE-based browsers accessing a complex web sites such as `microsoft.com`.

Detecting insecure component usage from the gmail workload is relatively more time consuming. In particular, the analysis of `MSHTML.dll` took about four hours. The main reasons are as follows. First, the user login is necessary to access the gmail account, and `MSHTML.dll` is heavily used for this [28]. Second, when an instruction is executed, the second phase in our framework checks whether or not the instruction is to be instrumented. Because `MSHTML.dll` is a large file whose size is about 12MB, a large number of checking is necessary, even though the number of instrumented instructions is relatively small. Despite the additional performance overhead, the analysis time for each browser is reasonable. For example, the analysis of `MSHTML.dll` used by IE Tab 2 took about thirty minutes.

It is interesting to note that the offline analysis phase for `MSHTML.dll` is relatively fast. This is because the size of the traces extracted is much smaller. For example, while the analysis on `URLMON.dll` generates runtime traces of 335MB during the gmail workload, the analysis on `MSHTML.dll` only generates runtime traces of 7.53 MB. The main reason is that the accesses to the global data region of `MSHTML.dll` is rare at runtime. For the gmail workload performed by IE Tab 2, `URLMON.dll` and `MSHTML.dll` access their global data regions 800,120 times and 1,585 times, respectively.

4.5 Further Discussion and Analysis

We now further discuss a natural question: Are the other detected inconsistent policy configurations security relevant? [3]

Our framework provides useful information on inconsistent configurations and evaluations of security polices. As we have discussed, such information is effective at detecting and analyzing insecure component usage. However, the detected inconsistent executions may not all be security relevant. For example, `WININET.dll` maintains a configuration of Autodial [2] in its global data region, and only IE enables it. Also, while IE initializes User-Agent String [48] whose configuration is stored in `URLMON.dll`, the test subjects do not. Although these configurations are inconsistent, they may not introduce security issues for the test subjects. However, such information can still be valuable for improving the functionalities of the client software using the components.

To determine the importance of the detected inconsistencies, domain knowledge of the test subjects is typically needed. Sometimes we are not able to determine whether the detected inconsistencies may cause security vulnerabilities. For example, an internal function of `URLMON.dll` takes the data stored in the global data region as a parameter, and a conditional jump is affected by the return value of the function. We observed that IE falls through and the test subjects take the branch at the conditional jump. Although this inconsistency may lead to a security problem, it is difficult to fully determine as we do not know the precise semantics of this function.

Also, the inconsistent policy configurations detected from the non-IE components (such as Adobe Flash Player and QuickTime Player) can also lead to security vulnerabilities. However, because we lack domain-expert knowledge on these components, we have focused our analysis on the IE components. In particular, the source code and the detailed documentation for the non-IE components are not publicly available. On the other hand, our results have clearly demonstrated that inconsistent policy configuration is a general concern that affects many applications.

5. RELATED WORK

This section surveys closely related work, which we divide into four categories: bug detection via inconsistent software behavior detection, detection of component insecurity, frameworks for secure component usage, and detection of violated browser policies.

Bug Detection via Inconsistent Software Behavior Detection. Brumley *et al.* [10] present a bug detection technique to discover deviations among different implementations of the same protocol specification. The technique is related to ours because it is also based on the general differential analysis concept. It analyzes different software and detects inconsistent behavior that is supposed to be consistent. However, the technique has a different goal from ours. Their goal is to detect inconsistent implementations of the same protocol, while ours is to detect inconsistent policy evaluations that may lead to insecure component reuse.

Detection of Component Insecurity. The detection of insecure software components has been actively studied. Neuhaus *et al.* [41] propose a technique that performs statistical analysis on vulnerability history; the function calls and the imports of each vulnerable component are utilized to characterize the corresponding vulnerabilities. Bandhakavi *et al.* [3] present a static analysis to detect information flow vulnerabilities in Firefox extensions. Dhawan *et al.* [13] dynamically track the execution of JavaScript extensions in Firefox to detect information flow violations. Guha *et al.* [20] statically check security of the browser extensions by using software verification techniques. In comparison, while these techniques detect the insecurity of target components, we focus on detecting *insecure usage* of the components. In particular, we model insecure usage of a component as inconsistent evaluations of security policies maintained by the component. With this model, we are able to, for example, detect and analyze browser components insecurely used by IE-based browsers (see Section 4).

Framework for Secure Component Usage. Because malicious or vulnerable components can introduce security problems to software, extensive research has been conducted on protecting software against them. For example, secure browsers [19, 45, 50] apply sandboxing techniques to protect them from crashed plugins. Grier *et al.* [18] present security policies to use browser plugins in a secure manner. Barth *et al.* [5] propose a technique to mitigate the damage caused by the exploitation of vulnerable extensions by designing least privilege, privilege separation, and strong isolation. Kirda *et al.* [34] detect malicious browser components by monitoring spyware-like behavior. These techniques aim at detecting and protecting against insecure execution of target components, while our purpose is to detect insecure usage of components.

Detection of Violated Browser Policies. A browser's security policy serves as a key part for safe web browsing. Thus, modern browsers support a number of policies to improve their security [9]. Based on this insight, many researchers [4, 6, 7, 8, 33] have focused on detecting violations of browser security policies. Although our framework also detects the violation of browser security policies (Section 4), its goal is different from that of these previous techniques. We aim at detecting policies incorrectly configured by insecure browser component usage. In contrast, the aforementioned techniques detect subversions of the enforced security policies.

6. CONCLUSION

We have presented an effective framework to detect and analyze insecure component usage. Our key idea is to detect inconsistent security policy configurations. Suppose that both a reference and a test subject use a component that maintains the configuration of a security policy. If they use the component in ways that make the policy inconsistently evaluated, the test subject can be vulnerable to attacks intended to be blocked by the policy. We model component usage relevant to the policy as memory access patterns and the conditional jumps affected by them. Based on this model, we have presented a program analysis technique to locate inconsistent policy configurations at runtime. Our evaluation results show that our technique is effective at detecting and analyzing insecure component usage. In particular, it detected inconsistent policy configurations of real-world applications and discovered several new security vulnerabilities of IE-based browsers. We have also shown that our framework can be used effectively to conduct detailed analysis of security vulnerabilities related to insecure component usage.

For future work, we would like to analyze insecure usage of other widely-used components. Our current implementation focuses on analyzing the global data region to detect component usage relevant to security policies. We plan to expand the work's scope by handling other types of non-thread specific memory regions (*e.g.*, the heap) to detect and analyze general inconsistent component usage.

[3] In the tech report [35], we also discuss results on Gecko- and WebKit-based browsers.

References

[1] About URL Security Zones. http://msdn.microsoft.com/en-us/library/ms537183(v=vs.85).aspx.

[2] Autodial. http://technet.microsoft.com/en-us/library/cc781180(W.10).aspx.

[3] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: vetting browser extensions for security vulnerabilities. In *Usenix Security*, 2010.

[4] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *SSP*, 2009.

[5] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting Browsers from Extension Vulnerabilities. In *NDSS*, 2009.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS*, 2008.

[7] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*, 52, June 2009.

[8] A. Barth, J. Weinberger, and D. Song. Cross-origin javascript capability leaks: detection, exploitation, and defense. In *Usenix Security*, 2009.

[9] Browser Security Handbook. http://code.google.com/p/browsersec/wiki/Main.

[10] D. Brumley, J. Caballero, Z. Liang, N. James, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Usenix Security*, 2007.

[11] CAPEC-104: Cross Zone Scripting. http://capec.mitre.org/data/definitions/104.html.

[12] P. T. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, 2000.

[13] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. *ACSAC*, 2009.

[14] Enigma Browser. http://www.suttondesigns.com/.

[15] Executable and Linkable Format (ELF). http://www.skyfree.org/linux/references/ELF_Format.pdf.

[16] K. M. Goertzel, T. Winograd, and B. A. Hamilton. Safety and security considerations for component-based engineering of software-intensive systems. Technical report, https://buildsecurityin.us-cert.gov/swa/downloads/NAVSEA-Composition-DRAFT-061110.pdf, 2010.

[17] Green Browser 5.8. http://greenbrowser.en.softonic.com/.

[18] C. Grier, S. T. King, and D. S. Wallach. How I learned to stop worrying and love plugins. In *Web 2.0 Security and Privacy*, 2009.

[19] C. Grier, S. Tang, and S. T. King. Secure Web browsing with the OP Web browser. In *SSP*, 2008.

[20] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *SSP*, 2011.

[21] IDA Pro Disassmelber. http://www.hex-rays.com/idapro/.

[22] IDAPython. http://code.google.com/p/idapython/.

[23] IE Cross-site Scripting Filter. http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/cross-site-scripting-filter.

[24] IE Tab 2. https://addons.mozilla.org/en-US/firefox/addon/ie-tab/.

[25] IE Tab for Chrome. http://www.chromeextensions.org/utilities/ie-tab/.

[26] IE Tab for Firefox. https://addons.mozilla.org/en-US/firefox/addon/ie-tab/.

[27] IInternetSecurityManager::ProcessUrlAction Method. http://msdn.microsoft.com/en-us/library/ms537136(v=vs.85).aspx.

[28] Internet Explorer Architecture. http://msdn.microsoft.com/en-us/library/aa741312(28v=vs.85)29.aspx.

[29] Internet Explorer does not support user names and passwords in Web site addresses (HTTP or HTTPS URLs) . http://support.microsoft.com/kb/834489.

[30] Internet Explorer Local Machine Zone Lockdown. http://technet.microsoft.com/en-us/library/cc782928(WS.10).aspx.

[31] Internet Explorer Zone Elevation Blocks. http://technet.microsoft.com/en-us/library/cc757200(WS.10).aspx.

[32] Introduction to Feature Controls. http://msdn.microsoft.com/en-us/library/ms537184(v=vs.85).aspx.

[33] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from dns rebinding attacks. In *CCS*, 2007.

[34] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *Usenix Security*, 2006.

[35] T. Kwon and Z. Su. Automated detection and analysis of insecure component usage. Technical report, http://www.cs.ucdavis.edu/research/tech-reports/2011/CSE-2012-25.pdf, 2012.

[36] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[37] Lunascape 6. http://www.lunascape.tv/.

[38] W. M. Mckeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[39] Microsoft Portable Executable and Common Object File Format Specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

[40] Naming Files, Paths, and Namespaces. http://msdn.microsoft.com/en-us/library/aa365247\%28v=vs.85\%29.aspx.

[41] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *CCS*, 2007.

[42] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS*, 2004.

[43] Slim Browser 5.01. http://www.slimbrowser.net/en/.

[44] SmartScreen Filter. http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/smartscreen-filter.

[45] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In *OSDI*, 2010.

[46] Top 10 Chrome Browser Add-ons. http://www.pcworld.com/article/185744/top_10_chrome_browser_addons.html.

[47] Trident (layout engine). http://en.wikipedia.org/wiki/Trident_(layout_engine).

[48] Understanding User-Agent Strings. http://msdn.microsoft.com/en-us/library/ms537503(v=vs.85).aspx.

[49] UTF-8. http://en.wikipedia.org/wiki/UTF-8.

[50] H. Wang, C. Grier, A. Moshchuk, S. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Usenix Security*, 2009.

[51] WebbIE 3.14. http://www.webbie.org.uk/.

[52] O. Whitehouse. GS and ASLR in Windows Vista. In *Black Hat DC*, 2007.

[53] You receive a "File Not Found" error message in Excel when you open a file by double-clicking the file name. http://support.microsoft.com/kb/207574.