Hidden GEMs: Automated Discovery of Access Control Vulnerabilities in Graphical User Interfaces

Collin Mulliner
Northeastern University
crm@ccs.neu.edu

William Robertson
Northeastern University
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
ek@ccs.neu.edu

***Abstract*—Graphical user interfaces (GUIs) are the predominant means by which users interact with modern programs. GUIs contain a number of common visual elements or widgets such as labels, textfields, buttons, and lists, and GUIs typically provide the ability to set attributes on these widgets to control their visibility, enabled status, and whether they are writable. While these attributes are extremely useful to provide visual cues to users to guide them through an application's GUI, they can also be misused for purposes they were not intended. In particular, in the context of GUI-based applications that include multiple privilege levels within the application, GUI element attributes are often misused as a mechanism for enforcing access control policies.**

In this work, we introduce *GEMs*, or instances of GUI element misuse, as a novel class of access control vulnerabilities in GUI-based applications. We present a classification of different GEMs that can arise through misuse of widget attributes, and describe a general algorithm for identifying and confirming the presence of GEMs in vulnerable applications. We then present GEM Miner, an implementation of our GEM analysis for the Windows platform. We evaluate GEM Miner over a test set of three complex, real-world GUI-based applications targeted at the small business and enterprise markets, and demonstrate the efficacy of our analysis by finding numerous previously unknown access control vulnerabilities in these applications. We have reported the vulnerabilities we discovered to the developers of each application, and in one case have received confirmation of the issue.

I. Introduction

Graphical user interfaces (GUIs) are the predominant means by which users interact with modern programs. GUIs were introduced because the use of visual elements such as icons and standard controls as well as pointing devices – e.g., the mouse – are much more intuitive for most users than command line interfaces. As a result, GUIs have become ubiquitous, and they are found on a wide range of computing devices (e.g., desktops, tablets, and mobile phones).

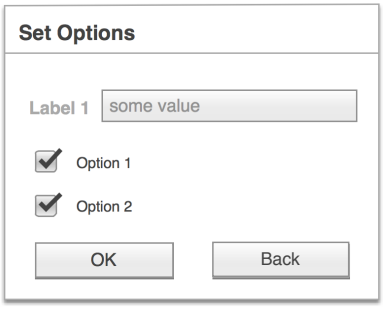
From a development perspective, the once-difficult process of creating GUI-driven applications has become relatively straightforward. Every major operating system includes and supports software tools and frameworks for the rapid development of GUI-based applications. For example, the popular Microsoft Foundation Class Library (also known as Microsoft Foundation Classes, or MFC) is a framework that wraps portions of the Windows API in C++ classes, including functionality that enables developers to quickly create applications with a standard look-and-feel. MFC classes are defined to wrap many of the low-level handle-managed Windows objects, and also provide numerous predefined windows and common controls.

In a typical GUI, user interface elements can be programmatically hidden, disabled, or made read-only through the manipulation of attributes on those elements. The idea of hiding UI components is to make it easier for the developer to instantiate and use these components when constructing the interface. That is, rather than deleting a user interface component and re-instantiating it every time it is needed, it can simply be hidden from view such that it is invisible to the user. Similarly, enabling and disabling UI components allows the developer to give visual cues to the user on the functionality that is available in the application. For example, if the user should not be able to press the "OK" button before filling out a textfield in a dialog, the button can simply be disabled and the user will not be able to press it. The GUI framework will ensure that the button cannot accept any mouse press events in the disabled state.

Although allowing user interface elements to have different attributes is useful as a feedback mechanism to the user, there is a caveat: Developers might start to misuse these attributes as an access control mechanism in the application logic. For example, a developer might disable a textfield if the user is not authorized to enter any input into the backend database via the user interface. Generally speaking, developers might start to rely on user interface element attributes to enforce privilege levels within the application. Unfortunately, these user interface attributes *are not suitable* as an access control mechanism. In fact, tools such as WinSpy++ [7] can be used to select, view, and modify the attributes of any window in the system, including the entire hierarchy of widgets those windows contain. Also, Microsoft offers a similar utility called Spy++ [17] that ships with Microsoft Visual Studio. Note that such an external modification of the user interface components is not only possible on the Windows platform, but is a general design property of many GUI frameworks (e.g., Java Swing, GTK).

In this paper, we first introduce a novel class of vulnerabilities that we refer to as *GEMs*, or instances of *GUI element misuse*. As discussed above, GEMs arise when developers rely on and misuse UI element attributes to implement access control checks, and to our knowledge have not been reported before in the literature. GEMs can be used as a basis for privilege escalation exploits and general access control bypass, and can be found across all operating systems and GUI frameworks. GEM-based privilege escalations in GUIs are highly critical since they require minimal effort and experience to exploit once discovered by the attacker. They can be used to

(a) The window embeds two labels, two textfields, and two buttons. One button is disabled.



(b) The window embeds one label, one read-only textfield, two checkboxes, and two buttons.

(c) The window embeds one list view, one checkbox, and two buttons. One button is disabled.

Fig. 1: Examples of graphical user interface elements.

easily circumvent UI-based access control mechanisms if the developer has not taken the necessary precautions to protect the application. Note that although simple, GEMs can be difficult to detect in complex applications. Hence, in the second part of the paper, we present GEM Miner, a system to automatically detect GEM vulnerabilities in application GUIs. We built a prototype system for the Windows platform and evaluated it using three real-world, complex commercial applications. Using GEM Miner, we were able to identify numerous previously unknown GEM vulnerabilities. We have notified the impacted software vendors, and one of them confirmed our findings.

This paper makes the following contributions:

- We introduce a novel class of vulnerabilities that we refer to as GEMs, or instances of GUI element misuse. GEMs are a pervasive class of security problems in GUI-based applications. To our knowledge, GEMs have not been discussed in the literature before as a class of access control vulnerabilities.
- We present a classification of GEM vulnerabilities, and describe a general algorithm for automatically identifying and confirming the presence of such vulnerabilities.
- We have created an implementation of our GEM analysis for the Windows platform called GEM Miner, and demonstrate its efficacy by evaluating it over three real-world commercial applications. Our evaluation shows that our GEM analysis is effective by identifying numerous previously unknown GEM vulnerabilities in these applications.

The rest of the paper is organized as follows. Section II motivates the problem and describes the threat model we assume. Section III presents a classification of GEMs. Section IV describes our algorithm for automatically mining GEMs in GUI-based applications. Section V describes our implementation of the GEM mining algorithm for the Windows platform. Section VI presents the results of applying our system to several real-world commercial applications. Section VII discusses related work, and Section VIII concludes the paper.

II. Background

In this section, we provide background information on graphical user interfaces, how they can be misused, and discuss our threat model for GEM vulnerabilities.

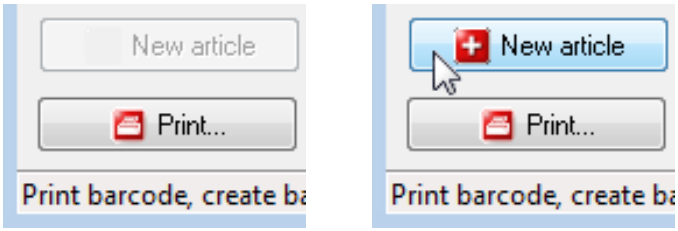
A. Graphical User Interfaces

A typical graphical user interface (GUI) is comprised of widgets. A widget is a self-contained visual element provided by a GUI framework to display data to the user, or to receive user input. Modern GUIs typically include common widgets such as windows, dialogs, pop-ups, buttons, textfields, checkboxes, and sliders that can be combined to build interactive, event-driven user interfaces.

Most operating systems now provide standard GUI frameworks that allow for the rapid development of GUI applications for that specific platform that share a common look-and-feel. Developers can often use drag and drop techniques to compose the user interface, and can additionally introduce custom widgets based on an underlying framework to extend its functionality.

In general, within a GUI system, only one widget can receive input at any given time. The exception to this are special events (e.g., special hot key sequences) that may be fired. Hence, such systems use the concept of *focus*. A widget that has the focus receives the user input such as key presses or mouse inputs. The focus can be changed through a number of different events such as moving the mouse over another widget, clicking a mouse button, or pressing a special key on the keyboard. Furthermore, the focus can also be changed programmatically.

In a typical GUI, user interface elements can be programmatically hidden, disabled, or enabled to make it easier to instantiate and use components when they are needed. This feature makes it easier to program GUI components and to selectively show parts of the user interface to the users according to the semantics of the application logic. For example, the button to submit a username and password might be programmatically disabled until both textfields have been filled by the user. Users have become conditioned to seeing



(a) The button is disabled to prevent user from creating a new article.

(b) The button now is enabled and can be selected and clicked.

Fig. 2: An example of a GEM in a real application. Here, we show that an attacker could enable the "New article" button so it can be clicked, even though the attacker should not have the ability to create new articles.

such visual cues, and the programmer can use these features to guide the user through the application's GUI. In contrast to enabled and disabled widgets that are visible to users, a hidden widget is still considered part of the UI, but it is not drawn to the screen and cannot receive any user input.

Note that operating systems might also provide more sophisticated widgets that have additional, advanced features. For example, widgets can be directly connected to database backends and can be used to automatically load data that is then displayed to the user. The textfields in the widget can then be set read-only to disallow modifications by the user.

Figure 1 shows three example windows that contain a number of widgets that should be familiar to all users.

B. Access control and GUIs

Enforcing access control is a basic requirement of any application that computes over sensitive data, which is often the case for applications targeted at the SMB or enterprise markets. Because GUIs offer the ability to set the visibility, enabled status, or writable status of UI components, a common assumption by many developers is that such components can safely be used as a part of the access control implementation of the application logic. After all, the operating system will ensure that the user is not able to press buttons that have been hidden or disabled, and that the user cannot simply type text into fields that are read-only. However, in reality, the GUI is not a reliable mechanism to enforce access control decisions because widgets in most GUI systems can simply be modified from outside of the application. That is, one can use standard OS services to interact with any UI component to inspect and modify the properties of any UI components that are running with the same OS-level privileges. Hence, if the application logic relies on UI components to enforce access control, such checks can easily be explored and bypassed by an attacker.

For example, consider the following simple GUI-based attack that is taken directly from one of the applications we analyzed in this work. The application in question is an inventory management application targeted at small businesses. The administrator has disabled the ability for lower-privileged users to create a new article in the inventory. Figure 2a shows a relevant part of the application's main window containing two buttons, where the button "New article" is disabled. Using a tool such as WinSpy++, an attacker can interact with the window and simply enable the button. Figure 2b shows the same button that is now enabled. The button can now be clicked by the attacker, allowing the attacker to create a new article in the database even though he is not authorized to access this functionality.

C. Threat Model

It is not uncommon for GUI applications to support different privilege levels within the application itself. In fact, applications in the enterprise context often process sensitive data that should only be accessible by specific users of the given system. These applications typically implement fine-grained access control schemes for reading and modifying the data processed by the application. This kind of access control is, again, highly application-specific, and limits the data and functionality to specific authorized users.

In our threat model, we assume that the application supports multiple privilege levels within the application that the attacker is interested in exploiting. For example, the application might support separate administrator and unprivileged user modes. In the user mode, certain privileged operations such as entering a new item into the database might be disabled, while in the administrator mode the full functionality of the application would be exposed to the user.

We also assume that the attacker has the ability to run programs on the target machine that can access the vulnerable application. This provides a large degree of power to the attacker, of course, since the attacker could in theory execute an exploit on the target machine to elevate her privileges, or reverse engineer the vulnerable application to either directly recover sensitive information or discover memory corruption vulnerabilities that could be exploited. However, we also assume that *the attacker is not necessarily technically sophisticated*. In the remainder of the paper, we demonstrate that GEM-based vulnerabilities can be successfully exploited by technically unsophisticated users by a process as simple as running a program to explore that GUI of the vulnerable program, a *significantly* lower bar than, for example, reverse engineering binary code (that may be obfuscated and that may be using anti-reversing techniques).

III. A Classification of GEMs

In this section, we present a definition of GEMs, or instances of GUI element misuse. We then enumerate several categories of GEMs that can arise in GUI-based applications.

We can model a GUI application as $A = \langle D, C, P, W \rangle$ where $D = \{d_1, d_2, \ldots\}$ is the set of data objects the application computes over, $C = \{c_1, c_2, \ldots\}$ is the set of event callbacks (or computations that are invoked in response to GUI events), $P = \{p_1, p_2, \ldots\}$ is the set of privilege levels users of the application can possess, and $W = \{w_1, w_2, \ldots\}$ is the set of widgets exposed in the GUI. Applications enforce an access control policy over D and C, which we model as

the functions read, write, and exec. Specifically we have,

$$\begin{aligned}\mathsf{read} &: \mathrm{D} \times \mathrm{P} \to \{\mathsf{true}, \mathsf{false}\} \\ \mathsf{write} &: \mathrm{D} \times \mathrm{P} \to \{\mathsf{true}, \mathsf{false}\} \\ \mathsf{exec} &: \mathrm{C} \times \mathrm{P} \to \{\mathsf{true}, \mathsf{false}\},\end{aligned}$$

where read determines whether a given privilege level is sufficient to read a data object, write determines whether a given privilege level is sufficient to modify a data object, and exec determines whether a given privilege level is sufficient to execute an event callback. Privileges can map to users of the application, roles assumed by users of the applications, or whether a user has obtained a license for the application, to name a few concrete examples.

To enforce the access control policy, the application must invoke a reference monitor that interposes on each type of operation over D and C. In many cases, the GUI framework itself is used by the application to implement this reference monitor. The main widget attributes relevant for policy enforcement are "enabled," "visible," and "value," which we model using the functions

$$\begin{aligned}\mathsf{enabled} &: W \times \mathrm{P} \to \{\mathsf{true}, \mathsf{false}\} \\ \mathsf{visible} &: W \times \mathrm{P} \to \{\mathsf{true}, \mathsf{false}\} \\ \mathsf{value} &: W \times \mathrm{P} \to \mathrm{D} \cup \{\varnothing\}.\end{aligned}$$

For instance, in the GUI-enforced monitor scheme,

$$\begin{aligned}\mathsf{read}\,(\mathrm{d_i}, \mathrm{p_j}) \equiv\ & \mathsf{visible}\,(w_\mathrm{k}, \mathrm{p_j}) \\ & \wedge \mathsf{value}\,(w_\mathrm{k}, \mathrm{p_j}) = \mathrm{d_i},\end{aligned}$$

which states that read access to $\mathrm{d_i}$ under privilege level $\mathrm{p_j}$ is granted if widget w_k contains the value $\mathrm{d_i}$ and w_k is visible under $\mathrm{p_j}$. Similarly, we have

$$\begin{aligned}\mathsf{write}\,(\mathrm{d_i}, \mathrm{p_j}) \equiv\ & \mathsf{enabled}\,(w_\mathrm{k}, \mathrm{p_j}) \\ & \wedge \mathsf{visible}\,(w_\mathrm{k}, \mathrm{p_j}) \\ & \wedge \mathsf{value}\,(w_\mathrm{k}, \mathrm{p_j}) = \mathrm{d_i} \\ \mathsf{exec}\,(\mathrm{c_i}, \mathrm{p_j}) \equiv\ & \mathsf{enabled}\,(\mathsf{widget}\,(\mathrm{c_i}), \mathrm{p_j}) \\ & \wedge \mathsf{visible}\,(\mathsf{widget}\,(\mathrm{c_i}), \mathrm{p_j})\end{aligned}$$

where $\mathsf{widget} : \mathrm{C} \to W$ maps event callbacks to the widgets they are registered with.

GEMs arise when the application developer in fact decides to rely upon the GUI framework itself to implement a subset – or all – of the reference monitor, since the required property of complete mediation is not actually provided by GUI frameworks. In particular, it is possible in most modern GUI frameworks for unprivileged users to directly modify the enabled, visible, and value attributes for all widgets, essentially granting complete control over the enabled, visible, and value relations.

Using this abstraction, we can enumerate several classes of GEMs.

1) Unauthorized information disclosure:

$$\text{(G1)} \qquad \mathsf{read}(\cdot) \not\equiv \mathsf{visible}\,(\cdot) \wedge \mathsf{value}\,(\cdot)$$

Information disclosure GEMs arise when applications enforce read access to data stored in a GUI element by setting the element's visibility attribute to false – i.e., an element containing sensitive data is simply hidden if the current user doesn't possess sufficient privilege to access that data. This situation can also arise if the application fails to scrub GUI elements that might contain sensitive data when switching between users with different privilege levels.

2) Unauthorized information modification:

$$\text{(G2)} \qquad \mathsf{write}(\cdot) \not\equiv \mathsf{enabled}\,(\cdot) \wedge \mathsf{visible}\,(\cdot) \wedge \mathsf{value}\,(\cdot)$$

Information modification GEMs occur when applications enforce write access to data stored in GUI elements via those elements' enabled and visibility attributes. This class of GEM can be useful for bypassing authentication procedures to gain elevated privileges.

One can further categorize information modification into *transient* and *persistent* modifications. Transient modifications exist for the duration of a session, while persistent modifications are saved to the application's backing store. Persistent modifications are often realizable in conjunction with the following class.

3) Unauthorized callback execution:

$$\text{(G3)} \qquad \mathsf{exec}(\cdot) \not\equiv \mathsf{enabled}\,(\cdot) \wedge \mathsf{visible}\,(\cdot)$$

Callback execution GEMs arise when the application enforces access to privileged event callbacks through the enable and visibility attributes of GUI elements. These elements are often buttons, but can also be invoked on focus or hover events generated by arbitrary elements. Privileged event callbacks can lead to persistent modifications of sensitive data or authentication bypasses, as two examples.

IV. The GEM Miner Analysis

Next, we present a generic algorithm for discovering GEMs in graphical user interfaces, a process we term *GEM mining*. The algorithm represents a black-box analysis of GUI-based applications to discover access control vulnerabilities and, therefore, does not assume visibility into the application code itself or any underlying data model. Additionally, the algorithm itself is agnostic of the specific graphical toolkit used by an application, and only assumes the ability to introspect on and modify widgets, features that are common to virtually all GUI frameworks in use today. We discuss our implementation of this algorithm for applications on the Windows platform in Section V, though we are confident that it is general enough to easily apply to other GUI frameworks.

GEM mining proceeds in four distinct phases: *a*) application seeding, *b*) UI exploration, *c*) GEM candidate identification, and *d*) GEM checking. In the seeding phase, the test engineer performs an initial configuration of the application under test (AUT), for instance by creating a number of users or roles at distinct privilege levels. In the second phase, human-assisted *UI exploration* is performed to recover the space of possible UI states for the AUT. *GEM candidate identification* is performed in the third phase, where the collection of UI states found during the exploration phase is analyzed to identify candidate access control vulnerabilities belonging to one or more of the GEM classes enumerated in Section III.

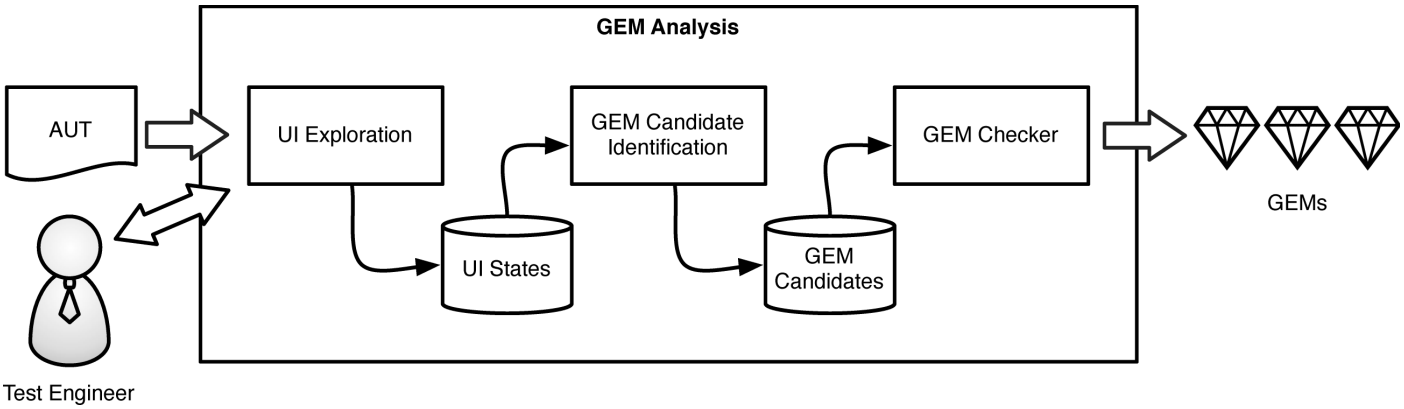


Fig. 3: Architectural overview of the GEM Miner analysis. The application is first seeded by a test engineer. Then, assisted UI exploration is performed to identify unique UI states. GEM candidate extraction examines the set of UI states found during the exploration. Candidate GEMs are confirmed in the final checking phase.

These candidates and the path through the GUI to reach them is analyzed in the final *GEM checking* phase to automatically confirm the vulnerability of the candidates. An overview of this process is presented in Figure 3.

In the remainder of this section, we elaborate on the details of each of these analysis phases.

A. Application Seeding

The first phase of the analysis is the application seeding phase. Seeding is generally required since applications often are not pre-configured with user accounts or roles at different privilege levels, a requirement for the remainder of the analysis. In addition, our analysis also requires data objects at different authorization levels in order to identify GEM candidates. As these considerations are application-dependent, we rely upon a test engineer to first familiarize themselves with the application's access control framework and prepare the application for testing. We enumerate common seeding activities in the following, though other steps could be required on a case-by-case basis.

1) Creating User Accounts or Roles: As mentioned previously, user accounts or roles with different privilege levels often need to be manually created by the test engineer as a first step of the seeding phase. The subjects might differ according to the design of the AUT; for instance, they could represent an unauthenticated user and administrator, an unlicensed user and a licensed user, or more generally a set of users with increasing levels of privilege. However, we require that at least two distinct privilege levels are available in the application – that is, $|P| \geqslant 2$. If two such levels do not exist in the application, then by our definition GEMs cannot exist in the AUT.

2) Creating Data: A second requirement of this phase is to seed the AUT with data. In particular, our analysis requires the presence of data with different authorization policies expressed in terms of the previously configured privilege levels. That is, given privilege levels $\{p_1, p_2, \ldots, p_n\}$, the test engineer should inject data objects $\{d_1, d_2, \ldots, p_m\}$ such that d_1 is readable and writable by subjects with privilege level p_1, d_2 is readable and writable by subjects with privilege level p_2, and so on.

```
1  for privilege_level in privilege_levels:
2      init_state = (initial_state(privilege_level), list())
3      state_queue = queue(init_state)
4      visited_states = set()
5
6      while not state_queue.empty():
7          state, path = state_queue.pop_front()
8          visited_states += state
9          actions = succ_actions(state)
10
11         for action in actions:
12             succ_state = apply_action(state, action)
13             if succ_state not in visited_states:
14                 succ_path = path + [ action ]
15                 state_queue += (succ_state, succ_path)
```

Fig. 4: UI state space exploration algorithm expressed as pseudocode.

Examples of data could include user profiles or items in a database, and is application-dependent.

3) Authentication Procedures: A final common requirement of the seeding phase is to record the steps required to authenticate and change privileges. Typically, this entails recording the sequence of UI actions required to reach and submit a login dialog, but is also generally speaking application-dependent.

B. UI Exploration

The second phase of the analysis explores the UI state space of the AUT. By UI state, we refer to the set of access control configurations for each widget w_i under privilege level p_j. More formally, we wish to recover all possible $s \in S$ such that

$$s = \langle \mathsf{enabled}(w_i, p_j), \; \mathsf{visible}(w_i, p_j), \; \mathsf{value}(w_i, p_j) \rangle \quad \forall w_i, p_j.$$

The algorithm our analysis uses to explore the UI state space is shown in Figure 4. For each distinct privilege level supported by the AUT and the seeding phase configuration, an

initial UI state of the application is extracted. This initial state is queued for exploration, and the visited state set is initialized to empty. Then, a breadth-first search is performed using the analysis queue for scheduling. At each iteration, the next state is retrieved from the queue and recorded as visited. The set of possible UI actions that can be executed on the given state is generated. Then, for each possible action, we apply the action and record the successor state and path of UI actions to reach that state. If this successor state is distinct from all previously visited states, then it is queued for analysis. Otherwise, it is discarded.

When generating a set of possible actions that can be applied to generate successor states, the exploration algorithm selects widgets from the current state that are likely to lead to the execution of an application event callback that will generate a new state. In practice, this amounts to recognizing UI elements such as buttons that often result in the creation of new windows or dialogs. The selection criteria, however, is a policy that can be modified to suit the needs of the AUT.

It is also important to note that during this phase, the analysis does not modify widget attributes – i.e., $\mathsf{enabled}(\cdot)$, $\mathsf{visible}(\cdot)$, or $\mathsf{value}(\cdot)$ – to avoid accidentally corrupting the application's state, which can lead to instability and program crashes.

UI state equality given two states s_i, s_j as used in the visited state set membership check is defined as equality between the $\mathsf{enabled}(\cdot), \mathsf{visible}(\cdot), \mathsf{value}(\cdot)$ values for the union of widgets present in both states at privilege level p_k.

We denote the set of UI states discovered in this phase as $\widehat{S} \subseteq S$. We note that we do not claim that the exploration algorithm we outline here is capable of retrieving *all* possible UI states, as reachability could be determined by factors other than those UI elements that are visible from the black-box perspective our analysis assumes. However, as we demonstrate in our evaluation in Section VI, this algorithm nevertheless performs well in practice.

During the exploration phase, a situation might arise where the analysis is unable to determine a valid action that will lead to a new successor UI state. To address this scenario, the analysis records a set of widgets that should lead to a new state and issues a request for assistance from the test engineer. The tester can manually inspect this set and provide feedback to the analysis in the form of UI actions that should be executed to produce new successors. The exploration is then restarted. We provide a concrete example of this feedback process in Section V.

C. GEM Candidate Identification

From the set of UI states $\widehat{S}$ collected during the exploration phase, our analysis then infers the access control policy the AUT intends to enforce. That is, from $\widehat{S}$, we recover read, write, and exec labels for each widget and privilege level pair (w_i, p_j).

Then, for each privilege level, the analysis compares the UI states reachable via a given path of UI actions executed during the exploration phase. This comparison is performed over all pairs of privilege levels p_i, p_j between each widget w_k and the attributes associated with that widget at each level. The identification phase then selects candidate GEMs based on the following criteria

$$\mathsf{visible}(w_k, p_i) \neq \mathsf{visible}(w_k, p_j)$$
$$\mathsf{enabled}(w_k, p_i) \neq \mathsf{enabled}(w_k, p_j).$$

Discrepancies between visibility attributes for the same widget at two privilege levels suggest the presence of an unauthorized information disclosure GEM (G1), while a discrepancy between enabled attributes suggests potential unauthorized modification (G2) or callback execution GEMs (G3). The set of candidate GEMs is recorded by the analysis as $\widehat{W} \subseteq W$.

D. GEM Checking

The final phase of the analysis is GEM checking, where the goal is to confirm candidates $\widehat{W}$ identified in the previous phase. Here, the analysis again drives the user interface of the AUT as in the exploration phase. However, in contrast to exploration, where the aim is to cover as much as possible of the UI state space, the checking phase replays the sequence of UI actions required to reach each candidate GEM that was recorded during exploration.

In the following, we describe the particular strategy used in the analysis to confirm each of the different classes of GEMs.

a) Unauthorized information modification (G2)*:* In confirming the presence of information modification GEMs, the general approach is, for each candidate widget $w_c \in \widehat{W}$, to replay the path of UI actions required to reach w_c under a privilege level p_i where $\mathsf{enabled}(w_c) = \mathsf{false}$. Then, w_c is enabled and a special value is injected into w_c by the analysis such that $\mathsf{value}(w_c) = x$. Next, the analysis attempts to persist the injected value by exercising possible successor actions to invoke application callbacks to that effect. Finally, the analysis replays the sequence of UI actions to reach w_c a second time and inspects it to obtain $\mathsf{value}'(w_c)$. If the injected value is still present – i.e., $\mathsf{value}(w_c) = \mathsf{value}'(w_c) = x$, then the presence of an information modification GEM has been confirmed.

The values that the analysis injects are drawn from different domains – e.g., integers, alphanumeric strings, floating point numbers – and different lengths in an attempt to automatically satisfy any input validation that the application might perform. Values from different domains are injected until one is accepted by the application, beginning with integers as they are also valid strings and are most often accepted. The entire checking sequence is repeated until an accepted value is found.

In addition, the procedure that we describe above is actually performed twice by the analysis to distinguish between *transient* and *persistent* modifications. In particular, to establish that a persistent modification GEM exists, the analysis closes the AUT after a potential successor callback has been invoked. Then, the application is restarted before replaying the UI action sequence to inspect w_c the second time. If the injected value is still present, then the analysis infers that the injected value was persisted to the backing store of the AUT, and concludes that a persistent modification GEM has been confirmed.

b) Unauthorized callback execution (G3)*:* Confirming the presence of callback execution GEMs is similar to the procedure for modification GEMs. For each candidate widget $w_c \in \widehat{W}$, the path to reach w_c is replayed under a privilege level p_i where $\mathsf{enabled}(w_c) = \mathsf{false}$. Then, w_c is enabled, set visible, and exercised; for instance, in the case of a button, a click event is generated.

In contrast to the modification confirmation case, the next step is to examine the successor UI state. If it is equal to the successor state of an invocation of w_c under privilege level p_j such that $p_i < p_j$ according to the definition of UI state equality given in the description of the exploration phase, then the analysis concludes that the invocation was successful and that a callback execution GEM has been confirmed.

c) Unauthorized information disclosure (G1)*:* In principle, the final class of GEMs that our analysis confirms should not require active checking of the AUT. Instead, this class of GEM should only require analysis of the set of UI states for the widget w_c across all possible privilege levels. In particular, our analysis would check – as it exactly does during the identification phase – whether there exists a pair of privileges p_j, p_k such that $p_j \neq p_k$ and $\mathsf{visible}(w_c, p_j) \neq \mathsf{visible}(w_c, p_k)$. If values were also present in w_c during the exploration phase, then the GEM could be considered confirmed.

However, in practice, this class of GEM exists in two flavors: disclosures based on widgets hidden in all UI states, and disclosures based on sensitive data that is "left over" in widgets after a higher-privileged user, such as an administrator, has interacted with the GUI. We term this subcategory of vulnerabilities *dangling* GEMs.

In cases where an application allows for temporary elevation of privileges, the order of privileges that the exploration is performed under becomes important. That is, in order to detect dangling GEMs, the analysis must first exercise the AUT as a high-privilege user, inject data into a candidate widget, and then lower privileges and check whether the data is still present in the widget. Therefore, to confirm dangling GEMs, our analysis performs the above procedure in this phase.

V. Implementation

Our prototype implementation of the GEM Miner analysis targets the Microsoft Windows platform, since it is heavily used in commercial environments to develop applications with complex access control schemes that are prone to GEMs. In this section, we provide background information on Windows GUI frameworks, present the prototype, and discuss practical aspects of its implementation.

A. Windows Applications

GUI applications on Windows consist of a hierarchy of widgets, where top-level widgets correspond primarily to windows and dialogs that contain sub-trees of widgets. Windows and dialogs typically contain one or more frames and layouts that dictate how groups of widgets are grouped and displayed to users. Widgets below the level of layout containers include familiar examples such as textfields, buttons, drop down lists, tables, checkboxes, and radio buttons.

On Windows, each widget has a number of standard attributes, including size, visibility, whether it is enabled, and whether it is writable. Windows does not directly define widget types, however. Typically, applications use widgets defined by higher-level GUI frameworks.[1]

B. Widget Types

The Windows API allows programs to query a widget's type, which is returned as a string. Due to variances between different GUI frameworks, a button might be labeled as a `Button`, `TBitBut`, or one of many other possibilities. Therefore, to achieve complete coverage of the possible GUI elements used on Windows, one would have to learn the entire set of all widgets that are provided by the many GUI frameworks used on the platform. For our prototype, however, we restricted our widget recognition logic to basic widget types such as buttons, checkboxes, and textfields. To accomplish this, we learned a mapping between widget names and types for a set of popular GUI frameworks.

Windows provides the ability to delegate widget rendering and logic to GUI frameworks, and many take advantage of this capability. Therefore, the low-level Windows API does not always provide the ability to query and manipulate certain widget sub-trees in applications under test. We denote such widgets as *opaque*. Common examples of opaque widgets include custom table widgets or button bars provided by high-level GUI frameworks.

C. Interacting with Widgets

Windows provides a wide range of APIs for programs to interact with widgets. One important set of functions are `SendMessage` and `SendMessageTimeout`, which allows programs to send events such as mouse clicks and key presses directly to a specific widget. Another useful function is `SendInput`, which allows programs to inject keyboard and mouse events into the Windows event queue. The difference between these two functions is that `SendMessage` targets specific widgets regardless of the current focus, while `SendInput` directs events at whatever widget is currently in focus. Therefore, `SendMessage` is the preferred function due to its precision, but our implementation falls back to `SendInput` for opaque widgets that cannot be directly accessed.

Windows also provides an API to modify the state of a widget. A widget can be enabled or disabled using `EnableWindow(Handle, Bool)`. A widget can be set visible or hidden using `SetWindowPos(Handle, 0, 0, 0, 0, flags)` by setting `flags` to either `SWP_SHOWWINDOW` or `SWP_HIDEWINDOW`.

Finally, values contained in widgets that display data can also be interacted with. For instance, in the case of textfields, data can be read by sending the `WM_GETTEXT` message using `SendMessage`, while textfield data can modified using `SendMessage` to emulate "select all" (`EM_SETSEL`) and "replace" operations (`EM_REPLACESEL`).

[1] We note that in Windows terminology, all widgets are actually referred to as "windows." In our description of our implementation, we distinguish between top-level windows and widgets contained in those windows for the sake of clarity.

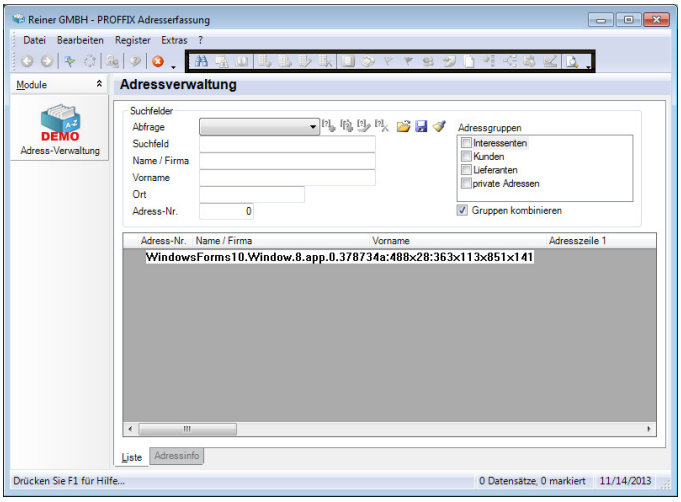


Fig. 5: Window with an annotated opaque widget. The widget ID is drawn at the center of the window.

Our implementation uses the Windows API function `EnumChildWindows` to enumerate all currently registered windows instantiated by the AUT. The UI state of each window is retrieved using `GetWindowLong`, `IsWindowVisible`, and `IsWindowEnabled`. The window type is determined using `GetClassName`.

D. User Interaction for Special Widgets

Each time the UI exploration encounters a widget that does not produce a new successor UI state, this fact is recorded. These facts are used to request assistance from the test engineer, who can then provide feedback in the form of an exception for the specific widget if necessary. Exceptions involve adding a specific offset inside the widget's bounding box to precisely target an injected mouse click. One example where this could be necessary is to click a specific button inside a button bar. Another case is when key press events must be used to set focus and execute a UI action inside of an opaque widget. For instance, we use this to select and activate an item in a table view by injecting an "arrow down" key press followed by a "return" key press.

To ease the process of creating exceptions, the implementation produces an annotated screen capture of UI states where new successors could not be identified. In particular, the problematic widget is highlighted by drawing a box around it. The test engineer can then easily identify and create an exception for the widget through manual inspection. Figure 5 shows an example of an annotated screen capture of Proffix.

The test engineer can further specify that a specific widget should be ignored by its ID, by its caption, or by its widget class. This capability is mostly needed to prevent undesired side effects during UI exploration, such as accidental print requests or modifications to the filesystem.

In our evaluation in Section VI, we quantify how many exceptions were required to analyze our case studies with our implementation.

E. Windows Widget Attributes

In addition to a widget's visibility and enabled status, Windows defines an extra attribute for some widgets such as textfields that controls their read-write status. For instance, on Windows, a textfield that is only used to display data needs to be enabled in order for the GUI to allow setting focus to the field. This could be used to allow the user to select the contents of the field in order to paste it elsewhere. However, since the field is only used for display purposes, it should not actually be editable. The read-write attribute is therefore provided to indicate that this is the desired state of the field.

While we do not explicitly model the read-write attribute in Section III, we note that for our purposes, the $\mathsf{enabled}(\cdot)$ relation subsumes both the enabled and read-write attributes found on Windows.

F. Detecting Dangling GEMs

Dangling GEMs are the result of "left over" data that is stored in widgets after a temporary switch from a lower privilege level to a higher privilege level within a single session. To detect this class of GEM, our implementation performs an additional step.

When the analysis determines that the UI exploration is complete, instead of simply terminating the AUT, the analysis records the UI state after dropping from a higher privilege level to a lower privilege level. Using this method, our implementation does not need to inject specific data into any widget to detect dangling information disclosure GEMs. Instead, it relies only on the fact that the AUT loads data into a privileged widget during its execution.

G. Generating Successor States

To generate new successor UI states, the implementation must indicate to the AUT that data entered into widgets is ready to be processed. This is usually accomplished by pressing a button; however, the correct subset of widgets to exercise is application-specific.

Our implementation uses a heuristic to select the appropriate widget to exercise. The heuristic we currently use extracts all buttons from the current UI state and compares their labels to a list of values that often suggest a computation leading to a new UI state will occur, such as "OK" or "Save." If no button is found, the implementation then iterates through all available buttons in the current UI state.

H. Limitations

Our prototype implementation of GEM Miner has some limitations that are important to discuss. First, opaque widgets that the implementation currently cannot interact with increase the time required to analyze applications, as the test engineer must provide feedback before the analysis can continue. This often occurs when an application uses complex or custom widgets such as lists, tables, and trees. We observe that support for many of these widgets could be added, but this is an extra implementation step that we have not yet completed. We also note that even without direct support for these widget types, our analysis is nevertheless able to confirm numerous GEM-based vulnerabilities as we demonstrate in Section VI. This is

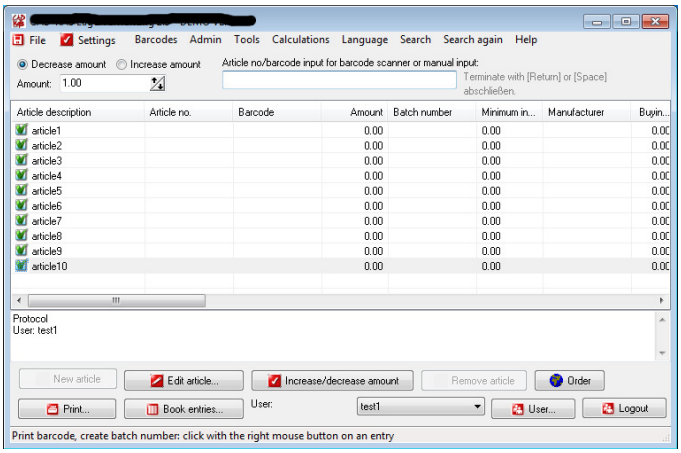


Fig. 6: The main window of App1.

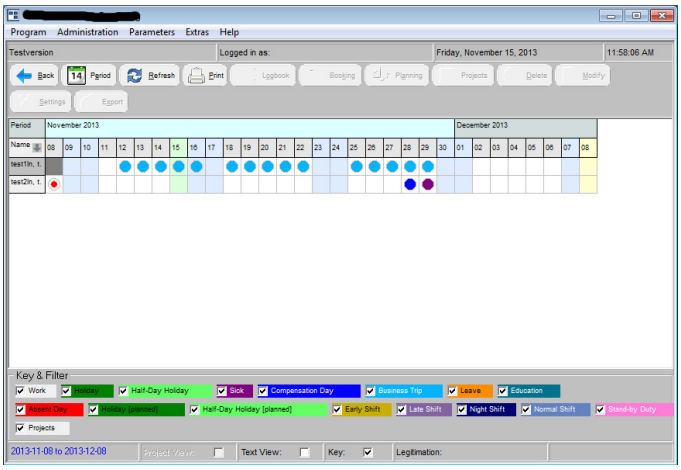


Fig. 7: The main window of App2.

due to our usage of mechanisms such as `SendInput`, which allows our implementation to inject events into the Windows event queue to the widget currently in focus, even if the implementation cannot directly interact with the widget.

Second, we do not claim that our exploration of the UI state space is complete. This would require an *a priori* specification of all possible widgets an application could instantiate, or extraction of this information from the program itself. Since, however, we use a black-box analysis, we do not attempt to extract this specification from AUTs. In addition, we note that full specifications of an application's UI do not exist in the general case, although some GUI frameworks do provide subsets of this information indirectly when GUI layouts are specified in a manifest accompanying the application (e.g., Android XML layouts). Regardless, our UI exploration technique, while incomplete, is nevertheless capable of covering most or all of the UI state space for real applications, and is capable of discovering a significant number of vulnerabilities for those applications.

VI. Evaluation

We evaluated GEM Miner, our implementation of the GEM mining analysis for the Windows platform, by applying

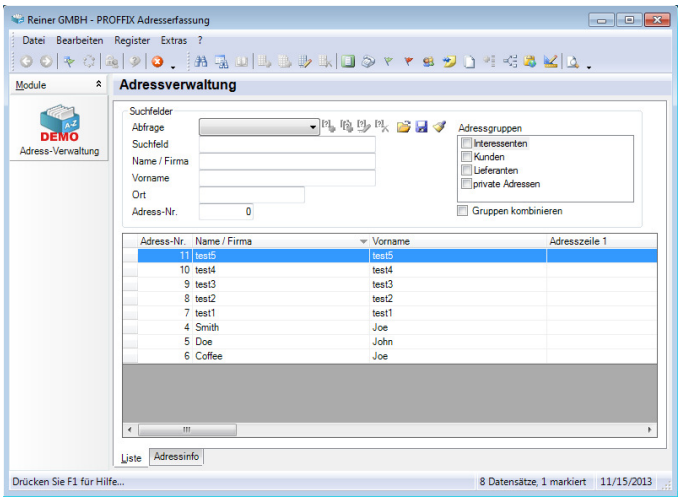


Fig. 8: The main window of Proffix.

it to three applications that incorporate multiple levels of privilege. Two of these were native applications written directly to the Win32 API, while the remaining one was written using the .NET framework. We selected the applications based on the fact that they implement multiple privilege levels within the application.

In the following, we first provide an overview of each of the applications under test by introducing the purpose of the applications, the access control schemes they implement, and how we seeded the applications for our tests. Then, we present the results of our analysis and discuss the results.

A. App1

App1 is an application for managing inventory (Figure 6). We analyzed the trial version of the software, while the full version of this software costs $55. The application supports multiple users, with the ability to grant or deny the ability to perform individual operations such as viewing, editing, or removing inventory items. The application also supports an administrative mode that grants full privileges to perform any operation provided by the application. The administrator mode is protected by password-based authentication.

The application requires a user to authenticate before it can be used. Switching to the administrative mode requires a user to log in prior to elevating privileges within the application.

To seed App1, we created one user account in addition to the default administrator role. For this less-privileged user, we added access control rules to deny the ability to perform two operations. We also added 10 articles to the inventory.

Tests conducted in high privilege mode were performed using the administrator role, while tests in low privilege mode were conducted as a user.

B. App2

App2 is an employee and project management application (Figure 7). We analyzed the trial version of this software in our evaluation; enabling the full version of the application requires

as $23/year subscription. The application supports multiple users and role-based privileges that can be assigned to a user account. Similarly to App1, this application also includes by default an administrator role that is protected by password-based authentication.

The administrator role has access to all functionality provided by the application. In addition, some operations are only available to the administrator. App2 does not require users to authenticate at application startup, but instead prompts users to authenticate at the time that a privileged operation is invoked during execution.

To seed the application, we created two user accounts and granted these accounts the ability to perform a subset of the full operations provided by the application. We did not need to supply additional seeding data, since the application provides employee management, and considers users of the application as employees.

Tests conducted in high privilege mode were performed as the administrator, while tests in low privilege mode were performed as the two users.

C. Proffix

Proffix [25] is an application that provides client management, order processing, and financial accounting functionality for small to medium-size businesses (Figure 8). The basic version of the software costs $2700. In addition, the functionality of Proffix can be extended for different tasks through a plugin architecture. Not all components of Proffix are available in the trial version, and so we focused our analysis on the address management component since it operates as a standalone application. The price of this component is $340/user.

Proffix supports multiple users and fine-grained access control for most aspects of the application. Similarly to the other two applications, an administrator account is included by default and possesses all privileges provided by the program. In particular, administrator privileges are required to create accounts for each individual user and grant specific privileges to each user. Privileges can be assigned based on software modules such as the address management component. Examples of assignable privileges available in Proffix include capabilities such as allowing read, write, and modify access to individual records and even fields inside individual records. The application requires users to authenticate at application startup.

We seeded Proffix with two user accounts that each have read access to the address database. Write access was only enabled for one of the users. We also added 10 address records to the database.

Tests conducted in high privilege mode were performed under the user account that had read and write access, while tests in low privilege mode were conducted as the user account with read-only access.

D. Analysis Time and Coverage

In this section of the evaluation, we report on the time required to analyze the applications in the test set, and the coverage of UI states achieved during the analysis using our GEM Miner implementation.

1) Analysis Time: Since the analysis we describe in this paper is human-assisted and not completely automated, giving a sense of the time required to analyze applications for GEMs must account for the time spent by the test engineer to provide feedback to the analysis as well as the time spent during the automated exploration phase. To that end, we report on the end-to-end analysis time in Figure 9a. This graph plots the number of UI states covered by the analysis against the total runtime of the analysis in minutes for each of the applications in the test set. Points on each line indicate restarts of the analysis due to requests for feedback from the test engineer to resolve ambiguities in generating successor UI states. Decreases in the number of UI states explored occur when undesired UI states are generated and blacklisted. Examples of this include when system dialogs like file save or print dialogs are instantiated as a result of UI actions.

To incorporate the time required for feedback, we used an estimate of two minutes for the average time required to provide feedback by a trained user of the implementation. This estimate is a conservative upper bound on the actual time required during the evaluation, which was usually much less.

In the case of Proffix, we observe that the analysis time required was just under 100 minutes to explore the entirety of the address book component of the application. This is in line with the fact that Proffix is a complex application with a highly-configurable access control scheme. During the initial period of the analysis, relatively few unique UI states are explored, as human feedback was required to blacklist problematic widgets and to instruct the explorer on how to make progress by finding valid successor states. However, after this initial period of around 18 minutes, the benefits of automation become clear as the implementation is able to quickly discover a large number of unique UI states. As the UI of the application is explored under different privilege levels, the rate of exploration slowly decreases until the entirety of the UI has been covered.

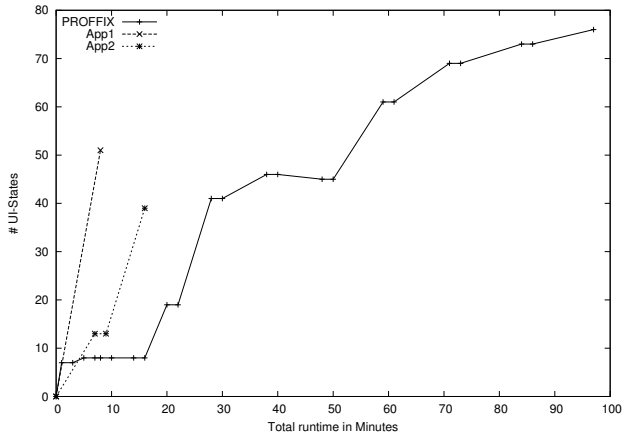
In the other two cases, exploration proceeded much more quickly as less feedback was required. We note that this is not necessarily solely a reflection of the complexity of the application itself, but is rather also due to the fact that much less feedback was required to cover all UI states of the application. In particular, App1 did not require any feedback from the test engineer at all in order to completely explore the application.

The subsequent phases of the analysis – i.e., GEM candidate identification and confirmation – are fully automatic and a function of the number of candidates identified. In our tests, these phases were completed on the order of a few seconds. We report on the end-to-end runtime required for all phases of the analysis over the test set in Table I.

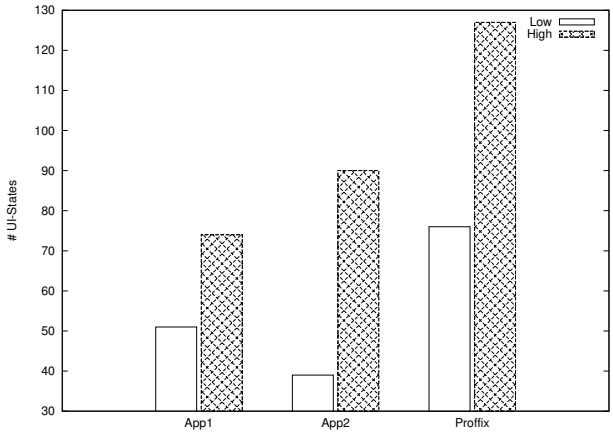
2) UI State Coverage: In Figure 9b, we show the absolute numbers of UI states discovered during low and high privilege explorations of each application. This demonstrates that each application contains a significant amount of functionality that is not available to low-privilege users.

E. GEM Detection

In our experiments, our prototype implementation of GEM Miner was able to discover and confirm a number of GEMs



(a) The graph shows the total time needed to explore each of our applications and the number of UI states explored at different points of the analysis. Every point represents a restart of the exploration component due to the incorporation of feedback from the test engineer. Decreases in the number of UI states occur when UI actions are blacklisted – for instance, when actions lead to the instantiation of file save or print dialogs. The App1 application is not restarted during the exploration, since it terminated without requiring feedback.



(b) Unique UI states discovered during the exploration of the test applications. The low privilege UI states correspond to states discovered either as a lower-privileged user, while high privilege UI states correspond to those discovered as an administrator.

Fig. 9: Time required for the analysis, taking into account test engineer feedback, as well as the number of unique UI states explored for each application in the test set in low and high privilege states.

in all applications in the test set. The GEMs that were found are drawn from all categories that we introduced in Section III – i.e., information disclosure (G1), information modification (G2), and callback execution (G3). An overview of the incidence of each category of GEM is shown in Table I, which further highlights the number of dangling information disclosures and distinguishes between transient and persistent modifications. GEM candidates that are not confirmed are widgets that can be manipulated, but that do not provide access to the target functionality. This can be the case due to some additional check in the application, not necessarily a security check. In the remainder of this section, we discuss the details of the particular GEMs we found in the context of each application.

1) Proffix: For the Proffix application, our analysis discovered 33 distinct GEM candidates. 23 of these were information modification GEMs in textfields or other editable widgets, while the remaining 10 were callback executions associated with button-style widgets.

Of the 23 modification GEMs, 17 were automatically confirmed during the checking phase. One additional modification candidate was confirmed as a mutation, where the application accepted the input but mutated the data prior to storing it. This mutation was due to the application performing input validation on input to the textfield. In particular, the textfield was intended to accept date entries, but performed an automatic conversion from integers to a valid date. Two additional modification candidates were manually confirmed where again, due to input validation, very specific input was required for confirmation. In these cases, input values were expected to be GPS coordinates.

Of the 10 callback GEMs, seven were automatically confirmed by the implementation, while one out of the remaining three was confirmed manually.

Since Proffix does not support temporary privilege escalation, there is no possibility for dangling data GEMs where sensitive data remains in UI elements after a temporary privilege escalation.

The GEMs discovered by the analysis enable attackers to modify data stored in the application's database that normal users are not authorized to, completely bypassing the access control scheme that the developers of the application intended.

2) App1: The analysis discovered two callback execution GEM candidates in the App1 application. The checking phase automatically confirmed both GEMs.

The analysis further discovered 44 information disclosure GEMs across a number of top-level windows. The design of the UI for the App1 application was unique in the test set, as the application creates a large number of windows, dialogs, and message boxes at application startup and simply shows or hides them as necessary during execution.

The callback GEMs identified during the analysis allows attackers to create new entries and delete existing entries in the application's database, despite the fact that the low privilege user was not authorized to do so. The disclosure GEMs, on the other hand, enable a wide range of attacks. One of the most critical of these is the fact that the account management window could be accessed by attackers, since it is created at startup and simply hidden from view. Therefore, an attacker could modify the visibility attribute on this window to directly access user credentials – i.e., usernames and passwords – and

	GEM Candidates			Automatically Confirmed			Manually Confirmed		
Application	**Disclosure**	**Modification**	**Callbacks**	**Disclosure**	**Modification**	**Callbacks**	**Modification**	**Callbacks**	**Runtime**
App1	44	-	2	44	-	2	-	-	51 sec
App2	1	1	8	-	-	4	-	2	205 sec
Proffix	-	23	10	-	17	7	3	1	666 sec
Total	45	24	20	44	17	13	3	3	

TABLE I: Statistics for GEM identification, automatic confirmation, manual confirmation, and total runtime for each application. Manually confirmed modification GEMs correspond to mutated inputs due to input validation.

log into the application. However, the administrator password was not included in this list.

The analysis also discovered one dangling information disclosure GEM in App1. In this case, when a user temporarily elevates privileges to administrator, the application prompts the user to provide the administrator password. After the user has dropped privileges, however, the administrator password remains in this authentication prompt. Therefore, by modifying the visibility attributes of that prompt, an attacker could recover the administrator password and gain full privileges to the application and its data. One concrete scenario for launching this attack would be for an authorized low privilege user to call a supervisor and ask her to perform an administrative task that requires elevation of privilege. After the supervisor has logged out, the low privilege user could then recover the administrator password left in the prompt.

3) App2: The analysis identified eight callback execution GEM candidates and one modification candidate in the App2 application. The checking phase automatically confirmed four callback GEMs, and manual testing confirmed an additional two callback GEMs. In the manual examination, we discovered that one of the unconfirmed callback GEM candidates was protected by an access control mechanism that checked if the administrator is logged in. If not, the application displayed a dialog with an error message explaining that the functionality is only accessible by the administrator. The write candidate, however, was neither confirmed during the checking phase nor through manual testing.

One of the callback GEMs allows modification of specific elements of all employee records. Another callback GEM provides the capability to export the entire employee database that contains recorded work time, scheduled business trips, and vacation days.

Finally, the analysis discovered one dangling information disclosure GEM in App2. The vulnerable widget is a list view element that holds the entire employee database, which includes all details of every employee. As with App1, this GEM requires a temporary elevation of privilege to the administrator role. When this occurs, the vulnerable widget is populated with the contents of the employee database. After administrator privileges are dropped, the widget still contains the employee data, which can be accessed by simply modifying the widget's visibility attribute.

F. Vulnerability Disclosure

All of the vulnerabilities discovered during the course of our experiments have been reported to the respective vendors. The developers of Proffix acknowledged our findings and indicated that they were unaware of the GEM issue. In particular, they were aware that widgets could be manipulated from outside the application, but were unaware of the security implications. They further indicated that they will specifically address the subject of GUI element misuse in their next major release. We have not received acknowledgment nor confirmation of the vulnerabilities we discovered in App1 and App2 from the developers of these applications. Therefore, we blinded the applications to prevent premature disclosure.

G. Countermeasures

We briefly investigated possible countermeasures against the attacks we describe in this paper. The most effective remediation would be to separate access control enforcement from the presentation layer – in effect, to introduce a proper reference monitor into the application logic. However, this could require substantial development time for legacy programs of any appreciable complexity. Instead of having developers build application-specific access control as part of their application logic, we anticipate that language-based policy specifications could be one scalable and secure mechanism for automatically implementing a reference monitor for GUI-based applications.

A partial solution that is perhaps less intrusive than retrofitting a full reference monitor would be to modify vulnerable applications to remove widgets that are not currently active instead of simply disabling or hiding them. This would be effective in the case of callback execution GEMs, but might not always be possible without a more invasive rewrite of the application, as in the case of a textfield that is used for modification of data in a high privilege mode and to display read-only data in a low privilege mode.

One possible lightweight countermeasure would be to run applications as a different operating system-level user. At least in the case of Windows, this would prevent the attacks from modifying the UI of vulnerable applications using the `SendMessage` functions. However, this countermeasure would only be effective if the local user does not have access to the application's OS user, and additionally that the local user does not have access to OS-level administrator privileges.

In general, application developers should not rely on the GUI to store runtime information and should rather treat widgets and the data stored within them as untrusted user input. Therefore, developers must implement at least basic input validation in the application's code that handles the GUI. Furthermore, to avoid data leaks developers should only create windows and widgets at the time a specific UI is needed.

Windows and widgets should be destroyed after they are no longer needed. In a client/server setting, access control must be implemented on the server side. Client-side access control should be treated as part of the application's usability component. It should only exist to guide the user, and to help him understand what an application considers to be well-formed input.

VII. Related Work

Concepts such as vulnerability testing, test case generation, fuzzing, and user interface testing are well-known in the software engineering and vulnerability analysis fields [3], [4], [11]. When analyzing web applications for vulnerabilities, black-box fuzzing tools [1], [8], [28] are often used due to their ease of use. Such scanners typically look for vulnerabilities such as cross-site scripting and SQL injections In addition to web-specific scanners, there exists a large number of more general vulnerability detection and security assessment tools. Most of these tools, such as Nessus [27] and Nikto [20], rely on a repository of known vulnerabilities that are tested. Other work [6], [5] in the area of security of web applications focused on client-side parameter tampering as a way to attack web applications. GEM Miner, in contrast, specifically focuses on the problem of GUI element misuse that, to our knowledge, has not been studied to date.

Besides application-level vulnerability scanners, there are also tools that operate at the network level, e.g., nmap [13]. These tools can determine the availability of hosts and services accessible on those hosts. However, they are not concerned with higher-level vulnerability analysis. There are also a large number of static source code analysis tools [14], [26], [29] that aim to identify vulnerabilities. These tools are orthogonal to GEM Miner.

A field that is closely related to our work is automated test case generation. The methods used to generate test cases can be generally summarized as random, specification-based [21], [23], and model-based [22] approaches. The well-known black-box fuzzing technique falls into the category of random test case generation. Similarly, we also generate test cases and analyze software for security-relevant bugs. However, we specifically look for GEM vulnerabilities that existing work does not address.

WinRunner [12], a well-known application testing tool, allows a human tester to record user actions such as key presses and mouse clicks and then replay these actions for testing. We note that the testing workflow WinRunner is designed for is not fully automated in terms of vulnerability discovery. The developer needs to write scripts and create checkpoints to compare the expected and actual outcomes from the test runs. Our test execution is similar to WinRunner in the sense that we also test for expected and actual outcomes. However, our approach automates vulnerability discovery and, in many cases, confirmation, and also is targeted specifically at UI-related privilege escalation vulnerabilities.

AutoIt [2], is a popular, free scripting language for Windows GUI automation. AutoIt works similarly to WinRunner, but is designed for GUI automation rather than GUI testing.

In software engineering, graphical user interface testing consists of the process of testing a software application's graphical user interface to ensure that it meets written specifications. Often, test cases are used to drive the testing process. A number of prior efforts have examined how such test cases can be automatically generated. Several techniques have been proposed to automatically generate test suites that are complete, and that simulate user behavior [16], [10], [15]. Note that none of the works in the domain of user interface testing has investigated the problem of GUI element misuse (GEM).

One relevant UI testing project is GUITAR, an automated system to test UI-driven applications for programming errors [19]. GUITAR consists of a GUI ripper and a number of test components. The GUI ripper is used to extract the UI states of an application, as well as the necessary UI interactions to reach each state. The test components works similarly to a fuzzer, permuting over the covered UI interactions to detect generic software errors such as application crashes. In comparison to GUITAR, our exploration component is similar to GUITAR's GUI ripper, but in addition to UI states, we also focus on security-relevant widget attributes such as visibility, enabled status, and writable flags. Additionally, our analysis focuses specifically on unauthorized privilege escalation in user interfaces in the form of GEMs, as opposed to more general program crashes.

Other relevant work investigates GUI testing using computer vision [9]. Here, a system has been built that uses computer vision to identify changes in the UI to determine if test cases succeeded or failed. This line of work is tailored towards functional testing of applications rather then finding security vulnerabilities. In our case, we cannot rely on visual observation since specific widgets attributes are not visually indicated.

We note that our checker component only shares very few similarities with traditional UI software testing components. The main commonality is that our approach automatically interacts with the UI of the target application. In contrast to existing techniques, however, our checker component is provided with a set of GEM candidates – i.e., potentially vulnerable widgets – the sequence of UI actions to reach these widgets, a test rule for detected widget types, and the expected results for a successful exploit. After the test is executed, the result is compared with the expected outcome and, if there is a match, the widget is reported to be a GEM vulnerability.

In 2002 and 2003, the Shatter attack was proposed [24], [18] that presented a new attack against Windows applications by exploiting the UI subsystem. This attack was based on removing the limits of textfields in the UI, and providing extra input as a part of a memory corruption vulnerability. In cases where the UI was running with elevated privileges (e.g., antivirus software), the attack could be used for OS-level privilege escalation. In response, Microsoft disabled UI manipulation across different OS-level user accounts. Note that in contrast to Shatter attacks, our approach does not manipulate the low-level internal bounds set in the implementations of UI elements in the OS. Rather, we focus on finding vulnerabilities that stem from developers blindly trusting UI elements for implementing access control logic in their applications.

VIII. Conclusion

In this work, we have introduced *GEMs*, or instances of GUI element misuse, as a new class of security vulnerability in GUI-based applications. GEMs arise when applications misuse the user interface as a mechanism for enforcing access control schemes when multiple privilege levels exist within the logic of an application.

We enumerated several classes of GEMs that can occur, and describe a general human-assisted technique that automates most of the process of discovering and confirming the presence of GEMs in applications.

We also built an implementation of this technique called GEM Miner to analyze applications targeting the Microsoft Windows platforms for GEMs. Our implementation demonstrates that it is possible to quickly discover exploitable security-relevant bugs in user interfaces with minimal effort on the part of test engineers. Using GEM Miner, we discovered and confirmed a number of previously-unknown GEMs in commercial Windows applications.

We view the work presented here as a first step towards automatic techniques for securing access control vulnerabilities in GUI-based applications. It is our hope that further work will improve upon GEM Miner, and that testing for GUI-based vulnerabilities will become a standard step in the application testing and security assessment process.

Acknowledgements

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0101. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work was also supported by the Office of Naval Research (ONR) under grant N000140911042, and by Secure Business Austria. The authors would like to thank Michael Weissbacher and Matthias Neugschwandtner for their insightful comments while writing this paper.

References

[1] Acunetix, "Acunetix Web Vulnerability Scanner," http://www.acunetix.com/, 2008.

[2] AutoItConsulting Ltd. (2013) AutoIt. http://www.autoitscript.com.

[3] B. Beizer, *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 1984.

[4] ——, *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[5] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan, "NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[6] P. Bisht, T. L. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan, "WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction." in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[7] J. Brown. (2012, July) WinSpy++. http://www.catch22.net/software/winspy-17.

[8] Burp Spider, "Web Application Security," http://portswigger.net/spider/, 2008.

[9] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI Testing Using Computer Vision," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2010.

[10] J. Clarke, "Automated test generation from a Behavioral Model," in *Proceedings of Pacific Northwest Software Quality Conference*, 1998.

[11] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Prentice-Hall International, 1994.

[12] HP/Mercury Interactive. (2003) WinRunner. http://support.openview.hp.com/encore/wr.jsp.

[13] Insecure.org, "NMap Network Scanner," http://www.insecure.org/nmap/, 2008.

[14] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)," in *IEEE Symposium on Security and Privacy (Oakland)*, 2006.

[15] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning," *IEEE Transactions on Software Engineering*, pp. 155–155, 2001.

[16] A. M. Memon, M. Pollack, and M. Soffa, "Using a Goal-driven Approach to Generate Test Cases for GUIs," in *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, 1999.

[17] Microsoft. (2013) Visual Studio Spy++. http://msdn.microsoft.com/en-us/library/dd460756.aspx.

[18] B. Moore. (2003, October) Shattering by Example. http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-moore/bh-us-04-moore-whitepaper.pdf.

[19] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, pp. 1–41, 2013.

[20] Nikto, "Web Server Scanner," http://www.cirt.net/code/nikto.shtml, 2008.

[21] J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," *Second International Conference on the Unified Modeling Language*, 1999.

[22] ——, "Using UML Collaboration Diagrams for Static Checking and Test Generation," *Third International Conference on UML*, 2000.

[23] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating Test Data from State-based Specifications," *Journal of Software Testing, Verification and Reliability*, 2003.

[24] C. Paget. (2002, August) Exploiting design flaws in the Win32 API for privilege escalation. http://www.thehackademy.net/madchat/vxdevl/papers/winsys/shatter.html.

[25] PROFFIX Software AG. (2013) PROFFIX. http://www.proffix.net.

[26] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," in *Symposium on Principles of Programming Languages*, 2006.

[27] Tenable Network Security, "Nessus Open Source Vulnerability Scanner Project," http://www.nessus.org/, 2008.

[28] "Web Application Attack and Audit Framework," http://w3af.sourceforge.net/.

[29] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," in *15th USENIX Security Symposium (USENIX SEC)*, 2006.