A Systematic Approach to Uncover Security Flaws in GUI Logic

Shuo Chen[†], José Meseguer[‡], Ralf Sasse[† ‡], Helen J. Wang[†], Yi-Min Wang[†]

[†] Systems and Networking Research Group
Microsoft Research
{shuochen,helenw,ymwang}@microsoft.com

[‡] Department of Computer Science
University of Illinois at Urbana-Champaign
{meseguer,rsasse}@cs.uiuc.edu

Abstract

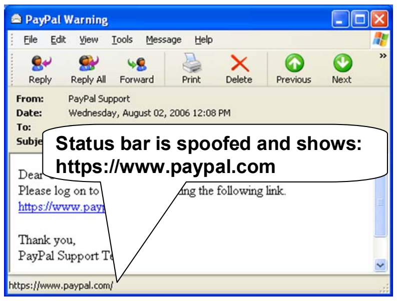
To achieve end-to-end security, traditional machine-to-machine security measures are insufficient if the integrity of the human-computer interface is compromised. GUI logic flaws *are a category of software vulnerabilities that result from logic bugs in GUI design/implementation. Visual spoofing attacks that exploit these flaws can lure even security-conscious users to perform unintended actions. The focus of this paper is to formulate the problem of GUI logic flaws and to develop a methodology for uncovering them in software implementations. Specifically, based on an in-depth study of key subsets of Internet Explorer (IE) browser source code, we have developed a formal model for the browser GUI logic and have applied formal reasoning to uncover new spoofing scenarios, including nine for status bar spoofing and four for address bar spoofing. The IE development team has confirmed all these scenarios and has fixed most of them in their latest build. Through this work, we demonstrate that a crucial subset of visual spoofing vulnerabilities originate from GUI logic flaws, which have a well-defined mathematical meaning allowing a systematic analysis.*

Keywords: Visual Spoofing, GUI Logic Flaw, Formal Methods, HTML, End-to-End Security

1. Introduction

Today, the trustworthiness of the web relies on the use of machine-to-machine security protocols (e.g., SSL or TLS) to provide authentication over the Internet to ensure that the client software (i.e., the browser) communicates with the intended server. However, such trustworthiness can be easily shattered by the last link between the client machine and its user. Indeed, the user-interface trust should be considered as a part of the *trusted path* problem in secure communications [7][8][25].

The exposure of the weakness between computer and human is not limited to non-technical social engineering attacks where naive users are fooled into clicking on an arbitrary hyperlink and download malicious executables without any security awareness. Even for a technology-savvy and security-conscious user, this last link can be spoofed visually. As shown in Figure 1(a), even if a user examines the status bar of the email client before she clicks on a hyperlink, she will not be able to tell that the status bar is spoofed and she will navigate to an unexpected website, instead of *https://www.paypal.com*. Furthermore, as shown in Figure 1(b), even if a user checks the correspondence between the URL displayed in the browser address bar and the top level web page content, she will not realize that the address bar is spoofed and the page comes from a malicious web site. Indeed, the combination of the email status bar spoofing and the browser address bar spoofing can give a rather "authentic" navigation experience to a faked PayPal page. Even SSL is not helpful – as shown in Figure 1(b), the spoofed page contains a valid PayPal certificate. Obviously, this can result in many bad consequences, such as identity theft (e.g. phishing), malware installation, and spreading of faked news.


PayPal Warning
File Edit View Tools Message Help
Reply Reply All Forward Print Delete Previous Next
Status bar is spoofed and shows: https://www.paypal.com

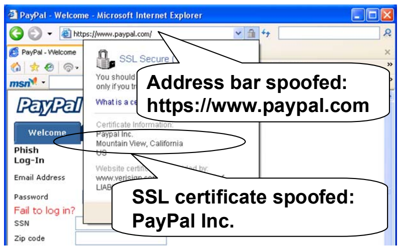
From: PayPal Support
Date: Wednesday, August 02, 2006 12:08 PM
To:

Dear
Please log on to ing the following link.
https://www.pay

Thank you,
PayPal Support T

https://www.paypal.com/

Figure 1(a): Status Bar Spoofing


PayPal - Welcome - Microsoft Internet Explorer
https://www.paypal.com/
SSL Secure
PayPal
Welcome
Phish Log-In
Email Address
Password
Fail to log in?
SSN
Zip code
Address bar spoofed: https://www.paypal.com
SSL certificate spoofed: PayPal Inc.

Certificate information
Paypal Inc.
Mountain View, California
US

Figure 1(b): Address Bar Spoofing

Visual spoofing attack is a generic term referring to any technique using a misleading GUI to gain trust from the user. Design/implementation flaws enabling such attacks are already a reality and have been sporadically discovered in commodity browsers

[21][22][23], including IE, Firefox, and Netscape Navigator. This paper focuses on a class of visual spoofing attacks that exploit *GUI logic flaws*, which are bugs in the GUI's design/implementation that allow the attacker to present incorrect information in parts of the authentic GUI that the user trusts, such as the email client status bar and the browser address bar. Figure 1(a) and (b) are just two instances of many such flaws that we discovered using the methodology described in this paper.

A second class of visual spoofing attack, which has been extensively discussed in previous research work [6][8][24][25], is to exploit graphical similarities. These attacks exploit picture-in-picture rendering [25] (i.e., a faked browser window drawn inside a real browser window), chromeless window (e.g., a window without the address bar or the status bar [8][25]), pop-up window covering the address bar, and symbol similarity (e.g., "1" vs. "l", "vv" vs. "w" [6], and non-English vs. English characters). We do not consider such attacks in this paper, but in Section 5 we briefly discuss how the graphical similarity problems are being addressed by researchers and browser vendors.

Our goal is to formulate the GUI logic problem and to develop a systematic methodology for uncovering logic flaws in GUI implementations. This is analogous to the body of work devoted to catching software implementation flaws, such as buffer overruns, data races, and deadlocks, through the means of static analysis or formal methods. Nevertheless, a unique challenge in finding GUI logic flaws is that these flaws are about what the user sees – user's vision and actions are integral parts of the spoofing attacks. Thus, the modeled system should include not only the GUI logic itself, but also how the user interacts with it.

In a nutshell, our methodology first requires mapping a visual invariant, such as "the URL that a user navigates to must be the same as that indicated on the status bar when the mouse hovers over an element in a static HTML page", to a well-defined program invariant, which is a Boolean condition about user state and software state. This mapping is done based on an in-depth understanding of the source code of the software. Our goal is then to discover all possible inputs to the software which can cause the visual invariant to be violated. In the example of finding status bar spoofing scenarios, we want to discover all HTML document tree structures that can cause the inconsistency between the URL indicated on the status bar and the URL that the browser is navigating to upon a click event; the resulting HTML tree structures can be used to craft instances of status bar spoofing attacks. To systematically derive these scenarios, we employ a formal reasoning tool to reason about the well-defined program invariant.

The methodology is applied to discover two classes of important GUI logic flaws in IE. The first class is the static-HTML-based *status-bar spoofing*. Flaws of this class are critical because static-HTML pages (i.e., pages without scripts) are considered safe to be rendered in email clients (e.g., Outlook[1] and Outlook Express) and to be hosted on blogging sites and social networking sites (e.g., *myspace.com*), and the status bar is the only trustworthy information source for the user to see the target of a hyperlink. The second class of flaws we studied is *address bar spoofing*, which allows a malicious website to hide its true URL and pretend to be a benign site. In both case studies, we use the Maude formal reasoning tool [2] to derive these spoofing scenarios, taking as input the browser GUI logic, program invariants, and user actions.

We have discovered nine canonical HTML tree structures leading to status bar spoofing and four scenarios of address bar spoofing. The IE development team has confirmed these scenarios and fixed eleven of them in the latest build, and scheduled to fix the remaining two in the next version. In addition to finding these flaws, we made the interesting observation that many classic programming errors, such as semantic composition errors, atomicity errors and race conditions are also manifested in the context of GUI implementation. More importantly, this paper demonstrates that GUI logic flaws can be expressed in well-defined Boolean invariants, so finding these flaws is done by inference about the violations of the invariants.

The rest of the paper is organized as follows. Section 2 gives an overview of our methodology. Sections 3 and 4 present case studies on status bar spoofing and address bar spoofing with IE. Section 5 presents discussions related to GUI security. Related work is given in Section 6. Section 7 concludes the paper.

2. Overview of Our Methodology

2.1 Our Analysis Approach

Figure 2 shows the major steps in our approach, based on formal analysis techniques. Existing formal analysis techniques have been successfully applied to reasoning about program invariants, e.g., the impossibility of buffer overrun in a program,

[1] Outlook does not show the target URL on the status bar, but on a small yellow tooltip near the mouse cursor. Because IE, Outlook and Outlook Express use the same HTML engine, most status bar spoofing scenarios can be transformed to email format to spoof Outlook tooltip and Outlook Express status bar.

guaranteed mutual exclusion in an algorithm, deadlock freedom in a concurrent system, secrecy in a cryptographic protocol, and so on. These program invariants have well-defined mathematical meaning. Uncovering GUI logic flaws, on the other hand, requires reasoning about what the user sees. The "invariant" in the user's vision does not have an immediately obvious mathematical meaning. For example, the visual invariant of the status bar is that if the user sees *foo.com* on the status bar before a mouse click, then the click must navigate to the *foo.com* page. It is important to *map such a visual invariant to a program invariant* in order to apply formal reasoning, which is shown as step (a) in Figure 2.

The mapping between a visual invariant and a program invariant is determined by the logic of the GUI implementation, e.g., a browser's logic for mouse handling and page loading. An in-depth understanding of the logic is crucial in deriving the program invariant. Towards this goal, we conducted an extensive study of the source code of the IE browser to *extract pseudo code to capture the logic* (shown as step (b)). In addition, we needed to explicitly *specify the "system state"* (shown as step (c)), including both the browser's internal state and possibly what the user memorizes. Steps (d) and (e) depict the formalization of *the user's action sequence* and *the execution context* as the inputs to the program logic. *The user's action sequence* is an important component in the GUI logic problem. For example, the user may move and click the mouse, or open a new page. Each action can change the system state. Another input to specify is the *execution context* of the system, e.g., a web page is an execution context for the mouse handling logic – the same logic and the same user action, when executed on different web pages, can produce different results.

When the user action sequence, the execution context, the program logic, the system state and the program invariant are formally specified on the reasoning engine, formal reasoning is performed to check if the user action sequence applied on the system running in the execution context violates the program invariant. Each discovered violation is output as a potential spoofing scenario, which consists of the user action sequence, the execution context and the inference steps leading to the violation. Finally, we manually map each potential spoofing scenario back to a real-world scenario (shown as step (f)). This involves an effort to construct a webpage that sets up the execution context and lures the user to perform the actions. The mappings (a)(b)(f) between the real world and the formal model are currently done manually, some of which require significant effort. In this paper, our contribution is mainly to formalize the GUI logic problem. Reducing the manual effort is future work.

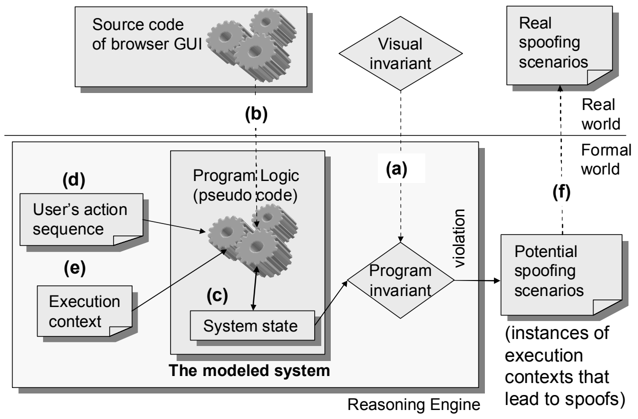


Figure 2: Overview of Our Methodology

2.2 Background: Formal Verification of Invariants in Maude

We formalize this problem within the logical framework of *rewriting logic* [12]. The corresponding reasoning engine is the Maude system [2]. In this paper, we use the term "Maude" to refer to both the Maude system and the language understood by it[2].

In Maude, the states of a system are represented by symbolic expressions, and the system transitions are specified by *rewrite rules* indicating how a state is transformed into another. For example, if we want to specify a 24-hour clock marking only the hours, we can declare a state constructor operator `clock` so that, say, `clock(7)` and `clock(21)` are two different clock states. In this example there is only one rewrite rule "ticking" the clock to the next hour. The clock system is specified as follows[3]:

```
type CLOCK .      var T : Int .
operator clock : Int -> CLOCK .
/*  This rule specifies the "ticking"  */
rule    clock(T)   =>    clock((T + 1) rem 24) .
```

where `Int` is the built-in data type of integers, a new type `CLOCK` of clock states is defined, and the state constructor `clock` is declared as an `operator` that takes an `Int` and produces a `CLOCK`. The clock "tick" transitions are specified by a rewrite rule introduced with the `rule` keyword, which rewrites a given clock marking time `T` to one marking time `((T+1) rem 24)`, that is, the remainder of `(T+1)` divided by 24. For example, `clock(23)` will be rewritten to `clock(0)`.

Once a system is specified, Maude's `search` command can be used to verify *invariants*. An invariant is a predicate that holds of an initial state and

[2] Similarly, people use the term "Perl" interchangeably for the Perl interpreter and the Perl language.

[3] In this paper, we use a syntax slightly different from Maude's.

of all states reachable from it. Suppose the initial state is `clock(0)`, and the invariant to verify is that the times it marks will always be greater than or equal to 0 and strictly smaller than 24. An invariant is verified by searching for any states violating it, i.e., for states satisfying the *negation* of the invariant. For our example, this can be done with the `search` command:

```
search clock(0)
=> clock(T) such that ((T < 0) or (T >= 24))
```

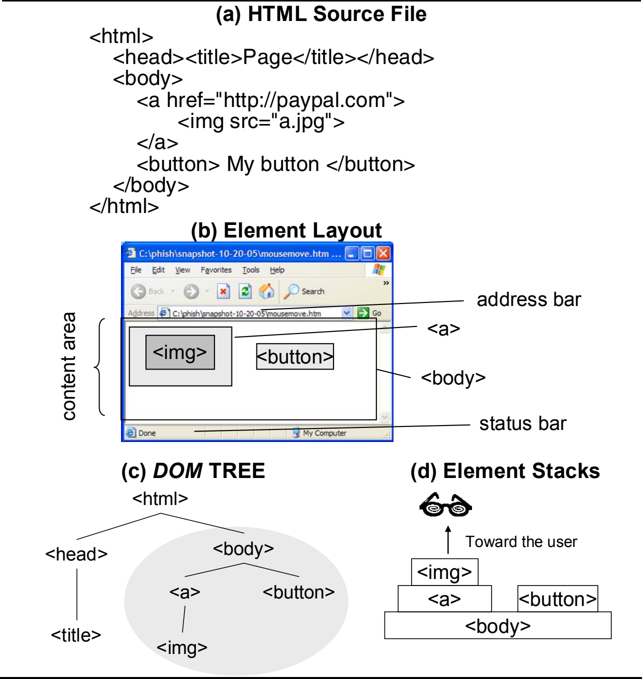
This search command responds: `No solution.` Therefore, the invariant is verified. In case an invariant is violated, the result will show a *trace* indicating the series of transitions leading to the violation. For a toy example like the one above, informal reasoning may convince us that a given invariant holds. But for complex situations, for example, the complex interactions between a user and a web browser, formal verification is needed in practice. This is exactly the way Maude is used in our work. As we explain in Sections 3.3 and 4.3, IE's status bar and address bar logics are specified by rewrite rules and equations in Maude, and the `search` command is used to search for spoofing scenarios.

3. Case Study 1: Status Bar Spoofing Based on Static HTML

Many web attacks, such as browser buffer overruns, cross-site scripting attacks, browser cross-frame attacks and phishing attacks, require the user to navigate to a malicious URL. Therefore, it is important for the user to know the target URL of a navigation, which is displayed on the status bar before the user clicks the mouse. Status bar spoofing is damaging if it can be constructed using only static HTML (i.e., without any active content such as JavaScript), because: (i) email clients, e.g., Outlook and Outlook Express, render static HTML contents only, and email is an important media to propagate malicious messages; (ii) blogging sites and social networking sites (e.g., *myspace.com*) usually sanitize user-posted contents to remove scripts, but allow static HTML contents.[4]

3.1 Background: Representation and Layout of an HTML Page

Background knowledge about HTML representation is a prerequisite for this case study. We give a brief tutorial here. An HTML page is represented as a tree structure, namely a *Document Object Model* tree, or *DOM tree*. Figure 3 shows an HTML source file, its DOM tree, and the layout of the page. The mapping from the source file (Figure 3(a)) to the DOM tree (Figure 3(c)) is straightforward – element *A* enclosing element *B* is represented by *A* being the parent of *B* in the DOM tree. The tree root is an `<html>` element, which has a `<head>` subtree and a `<body>` subtree. The `<body>` subtree is rendered in the browser's content area. Since status bar spoof is caused by user interactions with the content area, we focus on the `<body>` subtree in this case study.


(a) HTML Source File

```
<html>
  <head><title>Page</title></head>
  <body>
    <a href="http://paypal.com">
        <img src="a.jpg">
    </a>
    <button> My button </button>
  </body>
</html>
```

Figure 3: DOM Tree and Layout of an HTML Page

Figure 3(b) shows the layouts of elements from the user's viewpoint. In general, parent elements have larger layouts to contain children elements. Conceptually, these elements are stacked upwards (toward the user), with `<body>` sitting at the bottom (see Figure 3(d)). In HTML, `<a>` represents an anchor, and `<img>` represents an image.

3.2 Program Logic of Mouse Handling and Status Bar Behavior

Mouse handling logic plays an important role in status bar spoofs. We extracted the logic from the IE source code. It is presented here using pseudo code, which will be formalized in Section 3.3.

3.2.1 Central Logic

The mouse device can generate several raw messages. When a user moves the mouse onto a element and clicks on it, the sequence of raw messages consists of several MOUSEMOVEs, an

[4] A status bar spoof using a script is not a major security concern - it gets into a chicken-and-egg situation: a well-known site does not run an arbitrary script supplied from an arbitrary source. If the victim user has already been lure to in a malicious site, the goal of the spoofing has been achieved.

LBUTTONDOWN (i.e., left button down), and then a LBUTTONUP (i.e., left button up).

The core functions for mouse handling are `OnMouseMessage` and `SendMsgToElem`, which dispatch mouse messages to appropriate elements. Every element has its specific virtual functions `HandleMessage`, `DoClick` and `ClickAction` to implement the element's behaviors.

Each raw mouse message invokes an `OnMouseMessage` call (pseudo code shown in Table 1). The parameter `element` is the HTML element that is immediately under the mouse cursor. The parameter `message` is the type of the message, which can be either MOUSEMOVE, or LBUTTONDOWN, or LBUTTONUP. An `OnMouseMessage` call can potentially send three messages to HTML elements in the DOM tree: (i) if `element` is different from `elementLastMouseOver`, which is the element immediately under the mouse in the most recent `OnMouseMessage` call, then a MOUSELEAVE message is sent to `elementLastMouseOver`; (ii) the raw message itself (i.e., `message`) is sent to `element`; (iii) if `element` is different from `elementLastMouseOver`, a MOUSEOVER message is sent to `element`.

```
OnMouseMessage(element,message) {
  if (element != elementLastMouseOver)
        SendMsgToElem(MOUSELEAVE,
                      elementLastMouseOver)
  SendMsgToElem(message, element)
  if (element != elementLastMouseOver)
        SendMsgToElem(MOUSEOVER, element)
  elementLastMouseOver = element
}
```

```
SendMsgToElem(message,element) {
  btn = element.GetAncestor (BUTTON))
  if (btn != NULL && message == LBUTTONUP )
            element = btn
  repeat
      BubbleCanceled = loopElement
                  -> HandleMessage(message)
      loopElement = loopElement->parent
  until BubbleCanceled or loopElement is tree root
  if (message == LBUTTONUP)
     element->DoClick()  //handle the mouse click
}
```

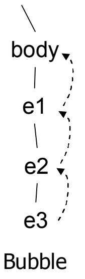


Table 1: `OnMouseMessage` and `SendMsgToElem`

In the function `SendMsgToElem()`, `btn` is the closest `Button` ancestor of `element`. If `btn` exists and `message` is LBUTTONUP (i.e., a click), then `element` becomes the button `btn`. It essentially means that any click on a descendant of a button is treated as a click on the button. Then, a *message bubbling* loop begins – starting from `element`, the virtual function `HandleMessage` of every element along the DOM tree path is invoked. Each `HandleMessage` call can cancel or continue the bubble (i.e., break out of or continue the loop) by setting a Boolean `BubbleCanceled`. After the bubbling loop, a mouse click is handled by calling the virtual function `DoClick` of `element`, when `message` is LBUTTONUP.

3.2.2 HTML Element Behaviors

An object class is implemented for each type of HTML element, such as `Anchor`, `Form`, `Button`, `InputField`, `Label`, `Image`, etc. These object classes inherit from the `AbstractElement` base class. The three virtual functions of `AbstractElement`, namely, `HandleMessage`, `DoClick` and `ClickAction`, implement default behaviors of real HTML elements. `AbstractElement::DoClick` (i.e., function `DoClick` of `AbstractElement)` implements a loop to invoke `ClickAction` of each element along the DOM tree path, similar to the bubbling in `SendMsgToElem`. `HandleMessage` and `ClickAction` of `AbstractElement` are basically "placeholders" – they simply return in order to continue the bubble.

Each HTML element class can override these virtual functions of `AbstractElement` to implement its specific behaviors. A subset of virtual functions of the `Anchor`, `Label` and `Image` elements is shown in Table 2. *These examples demonstrate the complexity in the mouse handling logic due to the intrinsic behavioral diversity of individual elements and the possible compositions*. For example, when the mouse is over an **anchor**, the target URL of this anchor will be displayed on the status bar by calling `SetStatusBar`, and the bubble continues, as indicated in `Anchor::HandleMessage`. When an anchor is clicked, `FollowHyperlink` is called to jump to the target URL, and the bubble is canceled, as indicated in `Anchor::ClickAction`. When the mouse is over a label, there is no `SetStatusBar` call, and the bubble is canceled. According to the HTML specification, a **label** can be associated with another element in the page, which is called "`ForElement`". Clicking on the label is equivalent to clicking on `ForElement`, as shown in `Label::ClickAction`. An **image** element can be associated with a *map*, which associates different screen regions on the image with different target URLs. When the mouse is over a region, the URL of the region is set to the status bar, as indicated in `Image::HandleMessage`. When the mouse clicks on the region, a `FollowHyperlink` call is made, as indicated in `Image::ClickAction`. If an image is not associated with a *map*, then the URL of the containing anchor of the image (i.e., the closest ancestor anchor of the image on the DOM) determines the status bar text and the hyperlink to follow.

Table 2: Virtual Functions of `Anchor`, `Label` and `Image` Elements

```
Bool Anchor::HandleMessage(message) {
   switch (message)
      case LBUTTONDOWN
         or LBUTTONUP:
            return true;     //cancel bubble
      case MOUSEOVER:
          SetStatusBar(targetURL)
          return false;    //continue bubble
      Other:
          return false;
 }

Bool Anchor::ClickAction() {
      FollowHyperlink(targetURL);
      return true;       // cancel bubble
}
```

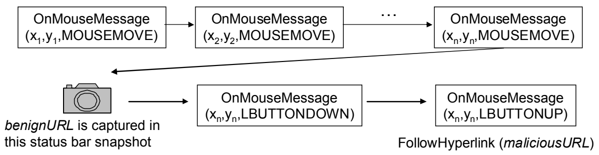
```
Bool Label::HandleMessage(message) {
   switch (message)
      case MOUSEOVER
         or MOUSELEAVE:
            return true; //cancel bubble
      Other:
          return false;
}

Bool Label::ClickAction() {
   forElement = GetForElement()
   if (forElement != NULL)
       forElement->DoClick();
   return true;
}
```

```
Bool Image::HandleMessage(message) {
   if a map is associated with this image
      MapTarget = GetTargetFromMap();
      switch (message)
         case MOUSEOVER:
            SetStatusBar(MapTarget)
            return true;
}
Bool Image::ClickAction() {
    if a Map is associated with this image
         MapTarget = GetTargetFromMap();
         FollowHyperlink(MapTarget);
    else pAnchor=GetContainingAnchor();
         pAnchor->ClickAction();
    return true;
}
```

3.3 Formalization of Status Bar Spoofing

The visual invariant of the status bar is intuitively that the target URL of a click must be identical to the URL displayed on the status bar when the user stops the mouse movement. The negation of this invariant defines a spoofing scenario (Figure 4): First, MOUSEMOVE messages on elements O_1, O_2, … , O_n invoke a sequence of `OnMouseMessage` calls. When the mouse stops moving, the user inspects the status bar and memorizes `benignURL`. Then, an LBUTTONDOWN and an LBUTTONUP messages are received, resulting in a `FollowHyperlink(maliciousURL)` call, where `maliciousURL` is different from `benignURL`.



OnMouseMessage (x_1,y_1,MOUSEMOVE)
OnMouseMessage (x_2,y_2,MOUSEMOVE)
…
OnMouseMessage (x_n,y_n,MOUSEMOVE)
benignURL is captured in this status bar snapshot
OnMouseMessage (x_n,y_n,LBUTTONDOWN)
OnMouseMessage (x_n,y_n,LBUTTONUP)
FollowHyperlink (*maliciousURL*)

Figure 4: Function Level View of Status Bar Spoof

We now apply the approach described in Figure 2.

(1) Specifying the user action sequence and the execution context (Steps (d) and (e) in Figure 2). A challenging question is how the spoofing possibilities can be systematically explored, given that the web page can be arbitrarily complex and the user's action sequence can be arbitrarily long. Canonicalization is a common form of abstraction used in formal reasoning practice to handle a complex problem space. For this particular problem, our goal is to map a set of user action sequences to a single *canonical action sequence*, and map a set of web pages to a single *canonical DOM tree*. Because any instance in the original problem space only trivially differs from its canonical form, we only need to explore the canonical state space to find all "representative" instances.

(1.1) Canonicalization of the user action sequence. In general the user action sequence consists of a number of mouse moves, followed by a status bar inspection, followed by a mouse click (button down and up). In a canonical action sequence, the number of mouse moves can be reduced to *two*. This is because, although each MOUSEMOVE can potentially update the status bar, the status bar is a memoryless object, which means: (i) upon every mouse action, how to update the status bar does not depend on any previous update, but only on the DOM tree branch corresponding to the current mouse coordinates; (ii) the whole sequence of status bar updates is equivalent to the last update. Thus, a canonical action sequence from element O1 to element O2 can be represented by the equation below, where the semicolon denotes sequential composition, and the MOUSEOVER on O1 invokes the last update of the status bar before the mouse arrives at O2 (O1 and O2 can be identical).

```
operator CanonicalActionSeq: Element Element -> ActionList .
equation  CanonicalActionSeq (O1,O2)
        = [   onMouseMessage(O1,MOUSEMOVE) ;
              onMouseMessage(O2,MOUSEMOVE) ;
              Inspection ;
              onMouseMessage(O2,LBUTTONDOWN);
              onMouseMessage(O2,LBUTTONUP) ] .
```

Note here that we use an equation instead of a rule. The difference between these is that an equation specifies a functional computation while a rule specifies a state transition.

(1.2) Canonicalization of the execution context (i.e., DOM trees). In general a DOM tree may have arbitrarily many branches, but we can restrict the number of branches of a canonical DOM tree to at most two. This is because the canonical action sequence contains at most two MOUSEMOVEs – the third branch of the DOM tree would be superfluous as it would not receive any mouse message. Each HTML element in the DOM tree is represented as an object with a unique identifier, a class, a parent attribute (specifying the DOM tree structure) and possibly other attributes. We currently model `Anchor`, `Button`, `Form`, `Image`, `InputField` and `Label` element classes, plus a `Body` element always at the root. For example, the term `< O | class:anchor,`

`parent:O'>` represents anchor element O whose parent is O'. Our analysis is restricted to canonical DOM trees of bounded size but sufficiently rich to uncover useful scenarios. Currently we have analyzed all one- and two-branch DOM trees with at most six elements. We also specify rules so that all canonical DOM trees satisfy the required HTML well-formedness restrictions, e.g., an anchor cannot be embedded in another anchor, an InputField can only be a leaf node, etc.

(2) Specifying system state and state transitions (Step (c) in Figure 2). *System State* includes the browser state `statusBar` and the user state `memorizedURL`. State transitions are triggered by the `SetStatusBar` action and the user's `Inspection` action as below, where `AL` is an arbitrary action list.

```
const Inspection : Action .
operator SetStatusBar : URL -> Action .
vars   AL : ActionList .    vars Url, Url' : URL .
rule   [SetStatusBar(Url) ; AL ] statusBar(Url')
   =>   [AL] statusBar(Url)  .
rule   [Inspection ; AL] statusBar(Url) memorizedURL(Url')
   => [AL] statusBar(Url) memorizedURL(Url)  .
```

The first rule specifies the semantics of `SetStatusBar(Url)`: if the current action list starts with a `SetStatusBar(Url)` action, and the status bar displays `Url'`, then after this action is completed, it disappears from the action list, and the status bar is updated to `Url`. The second rule specifies the `Inspection` action: if `statusBar` displays `Url`, the `memorizedURL` is an arbitrary value `Url'`, and the action list starts with `Inspection`, then after the inspection is made, `Inspection` disappears from the action list, and the URL on the status bar is copied to the user's memory, i.e., `memorizedURL`.

(3) Modeling the program logic (Step (b) in Figure 2). Modeling the functions shown in Table 1 and Table 2 is straightforward using Maude, e.g., `HandleMessage` and `ClickAction` of the `Anchor` element are specified in Table 3. Other functions are modeled in a similarly manner.

Table 3: Rules to specify `HandleMessage` and `ClickAction` of `Anchor`

```
equation  [AnchorHandleMessage(O,M) ; AL]   /* equation 1 */
   =  [cancelBubble ; AL]
         if M == LBUTTONUP or M == LBUTTONDOWN .
rule [AnchorHandleMessage(O,M); AL] <O |targetURL: Url...>
  => [SetStatusBar(Url) ; AL]  < O | targetURL: Url  >
      if M == MOUSEOVER .                         /* rule 2 */
equation   [AnchorHandleMessage(O,M) ; AL]  /* equation 3 */
   =   [no-op ; AL]
     if M • LBUTTONUP, LBUTTONDOWN or MOUSEOVER .
rule  [AnchorClickAction(O) ; AL]    < O | targetURL: Url  ... >
  => [FollowHyperlink(Url) ; cancelBubble ; AL]
        < O | targetURL: Url , ... > .            /* rule 4 */
```

It is easy to verify that these rules and equations indeed faithfully specify the behaviors of an `anchor` shown in Table 1: Equation 1 specifies that if an action list starts with an `AnchorHandleMessage(M,O)` action, this action should rewrite to a `cancelBubble`, if `M` is LBUTTONUP or LBUTTONDOWN. Rule 2 specifies that `AnchorHandleMessage(M,O)` should rewrite to `SetStatusBar(Url)` when handling MOUSEOVER, where `Url` is the target URL of the anchor. For any other type of message `M`, `AnchorHandleMessage(M,O)` should rewrite to `no-op` to continue the bubble, which is specified by equation 3. Rule 4 rewrites `AnchorClickAction (O)` to the concatenation of `FollowHyperlink (Url)` and `cancelBubble`, where `Url` is the target URL of the anchor.

(4) Specifying the program invariant (Step (a) in Figure 2). The only remaining question is how to define the negation of the program invariant to find status bar spoofs. It is specified as the pattern searched for in the `search` command:

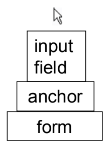
```
const maliciousUrl , benignUrl , empty : URL .
vars O1, O2: Element   Url: URL   AL: ActionList .
search CanonicalActionSeq(O1,O2)
            statusBar(empty)   memorizedUrl(empty)
      => [FollowHyperlink(maliciousUrl) ; AL]
            statusBar(Url)   memorizedUrl(benignUrl) .
```

The command gives a well-defined mathematical meaning to status bar spoofing scenarios: "the Maude initial term `CanonicalActionSeq(O1,O2) statusBar(empty) memorizedUrl(empty)` can be rewritten to the term `[FollowHyperlink (maliciousUrl) ; AL] statusBar(Url) memorizedUrl(benignUrl)`", which indicates that the user memorizes `benignURL`, but `FollowHyperlink(maliciousUrl)` is the next action to be performed by the browser.

3.4 Scenarios Suggested by the Results

We found nine combinations of canonical DOM trees and user action sequences that resulted in violations of the program invariant. All are due to unintended compositions of multiple HTML elements features. This section presents four representative scenarios in detail.

Shown in Figure 5, scenario 1 has an `InputField` embedded in an `anchor`, and the `anchor` is embedded in a `form`.

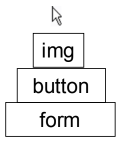


```
<form action="http://foo.com/" >
  <a href="http://paypal.com">
    <input type="image" src="faked.jpg">
  </a>
</form>
```

Figure 5: Illustration of Scenario 1

When the mouse is over the `InputField`, the `HandleMessage` of each element is called to handle the MOUSEOVER message that bubbles up to the DOM tree root. Only the anchor's `HandleMessage` writes its target URL *paypal.com* to the status bar, but when the `InputField` is clicked, its `ClickAction` method retrieves the target URL from the `form` element, which is *foo.com*. This scenario indicates the flaw in message bubbling – the MOUSEOVER bubbles up to the `anchor`, but the click is directly passed from the `InputField` to the `form`, skipping the `anchor`.

Scenario 2 (Figure 6) is very different from scenario 1: an `img` (i.e., image) associated with a map `ppl` is on top of a button. The target URL of `ppl` is set to *paypal.com*. When `img` gets a MOUSEOVER, it sets the status bar to *paypal.com* and cancels the bubble. When the mouse is clicked on `img`, because `img` is a child of `button`, the click is treated as a click on the button, according to the implementation of `SendMsgToElem()`. The button click, of course, leads to a navigation to *foo.com*. This scenario indicates a design flaw – an element (e.g., button) can hijack the click from its child, but it does not hijack the MOUSEOVER message, and thus causes the inconsistency.



```
<form action="http://foo.com/" >
  <button type=submit>
    <img src="faked_link.jpg" USEMAP= "ppl">
  </button>
</form>
<map name="ppl"><area href="http://paypal.com">
</map>
```

Figure 6: Illustration of Scenario 2

Scenario 3 contains a label embedded in an `anchor` (Figure 7(a)). When the mouse is moved toward the `label`, it must first pass over the anchor, and thus sets *paypal.com* on the status bar. When the `label` is clicked, the page is navigated to *foo.com*, because the `label` is associated with an anchor of *foo.com*. An opposite scenario shown as scenario 4 in Figure 7(b) seems more surprising, which suggests an outward mouse movement from a child to a parent. Such a movement makes it feasible to spoof the status bar using an `img` sitting on top of a `label`. Note that, because HTML syntax only allows an `img` to be a leaf node, such an outward mouse movement, which is suggested by Maude, is critical in the spoofing attack.

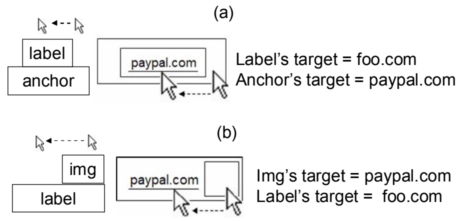


Figure 7: (a) Scenario 3 and (b) Scenario 4

We also derived several scenarios with two-branch DOM trees. They demonstrate the varieties of DOM trees and layout arrangements that can be utilized in spoofing, e.g., a spoof page places the two leafs side-by-side, another page uses *Cascading Style Sheets (CSS)* [16] to set element positions, etc.

4. Case Study 2: Address Bar Spoofing

Address bar spoofing is another category of spoofing attack. It fools users into trusting the current page when it comes from an untrusted source. The combination of a status bar spoofing and an address bar spoofing gives an end-to-end scenario to hide the identity of the malicious site, and thus is a serious security threat. In this section, we first introduce the background knowledge about the address bar logic, then present the Maude-based analysis technique and real spoofing scenarios uncovered by the analysis.

4.1 Background: Address Bar Basics

An IE process can create multiple *browsers*. Each one is implemented as a thread. A *browser*, built on the OLE framework [17], is a container (including the title bar, the address bar, the status bar, etc) hosting a client document in the content area. Many types of client documents can be hosted in IE, such as HTML, Microsoft Word, Macromedia Flash and PDF. The object to represent an HTML document is called a *renderer*. A *renderer* can host multiple *frames*, each displaying an HTML page downloaded from a URL. An HTML page is stored as a *markup* data structure. A *markup* consists of the URL and the DOM tree of the content from the URL. The top level *frame*, i.e., the one associated with the entire content area, is called the *primaryFrame* of the *renderer*. Figure 8 shows a *browser* displaying a page from *http://MySite*. The *renderer* has three *frames* – *PrimaryFrame* from *MySite*, *Frame1* from *PayPal.com* and *Frame2* from *MSN.com*. Each *frame* is associated with a *current markup* and, at the navigation time, a *pending markup*. Upon navigation completion, the pending markup is switched in and becomes the current markup.

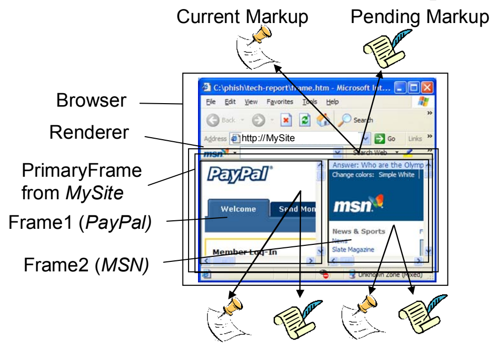


Figure 8: *Browser, Renderer, Frames* and *Markups*

Informally, the program invariant of the address bar correctness is that: (1) *the content area is rendered* *according to the current markup of primaryFrame;* and (2) *the URL on the address bar is the URL of the current markup of primaryFrame*. In the example shown in Figure 8, the address bar should display "*http://MySite*".

4.2 Overview of the HTML Navigation Logic

HTML navigation consists of multiple tasks – loading HTML content, switching markup, completing navigation and rendering the page. A *renderer* has an event queue to schedule these tasks. The event queue is a crucial mechanism for handling events asynchronously, so that the browser is not blocked to wait for the completion of the entire navigation. We studied three types of navigation: (1) loading a page into the current *renderer*; (2) traveling in the history of the current *renderer*; (3) opening a page in a new *renderer*. Due to space constraints, Figure 9 only illustrates a small subset of functions involved in the navigations.

Figure 9(a) shows the event sequence of loading a page in the current *renderer*. It is initiated by a `FollowHyperlink`, which posts a *start navigation* event. Function `PostMan` is responsible for downloading the new HTML content to a pending markup. Event *ready* is posted to invoke `SetInteractive`, to make the downloaded contents effective. `SetInteractive` first invokes `SwitchMarkup` to replace the current markup with the pending markup, and calls `NavigationComplete`. If the downloaded markup belongs to *primaryFrame*, function `SetAddressBar` is invoked to update its address bar. An `Ensure` event is posted by `SwitchMarkup`, which invokes `EnsureView` to construct a *View* structure containing element layouts derived from the current markup of *primaryFrame*. The OS periodically posts an *OnPaint* event to paint the content area by calling `RenderView`. Figure 9(b) shows the event sequence of a history travel. `History_Back` and `Travel` look up a history log to initialize the navigation. `PostMan`, in this case, loads HTML contents from a persistent storage in the hard disk, rather than from the Internet. The remaining portion of the sequence is similar to that of Figure 9(a).

Figure 9(c) shows the event sequence of loading a new page into a new *renderer*. `WindowOpen` is the starting point. It calls `CreatePendingDocObject` to create a new *renderer* and then call `SetClientSite`. `SetClientSite` prepares a number of Boolean flags as the properties of the new *renderer*, and calls `InitDocHost` to associate the *renderer* with the browser (i.e., the container). The new *renderer* at this moment is still empty. The `start-loading` event invokes `LoadDocument` which first calls `SetAddressBar` to set the address bar and then calls `Load` which calls `LoadFromInfo`. `CreateMarkup` and `SwitchMarkup` are called from `LoadFromInfo` before posting a `download-content` event to download the actual content for the newly created markup. Function `PostMan` does the downloading as above. The remainder of the sequence is similar to both prior sequences.

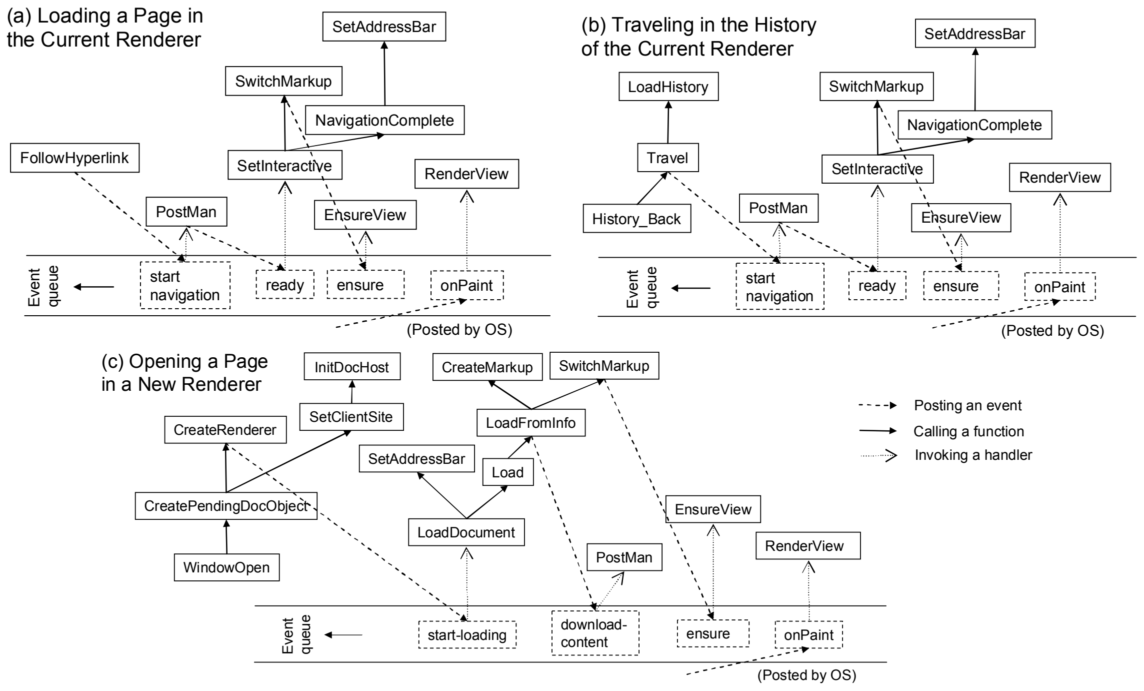


Figure 9: Logic of HTML Navigations

4.3 Formalization of the Navigations and the Address Bar Behavior

(1) Modeling the system state (Step (c) in Figure 2). Because an address bar spoofing is by definition the inconsistency between the address bar and the content area of the same browser, "spoofability" is a property of the logic of a single browser. This does not mean that only one browser is allowed in a spoofing scenario – there can be other browsers that create a hostile execution context to trigger a logic flaw in one particular browser. Nevertheless, we only need to model the system as one browser and prove its logical correctness (or uncover its flaws), and treat the overall effect of other browsers as the context of this browser.

The system state of a browser includes the URL displayed in the address bar, the URL of the *View* in the content area, a travel log and the primary frame. The Maude specification defines a set of *Frames* and a set of *Markups*. For example, if *Markup* `m1` is downloaded from URL `u1`, and it is the *currentMarkup* of *Frame* `f1`, we specify `f1` and `u1` as:

```
<f1 | currentMarkup: m1, pendingMarkup: …> <m1 |
URL: u1, frame: f1, …>
```

The system state also includes a function call queue and an event queue. The function call queue is denoted as $[\texttt{call}_1 ; \texttt{call}_2 ; \ldots ; \texttt{call}_n]$, and the event queue is denoted as $\{\texttt{event}_1 ; \texttt{event}_2 ; \ldots ; \texttt{event}_n\}$.

(2) Specifying the user action sequence (Step (d) in Figure 2). In the scenario of an address bar spoofing, the user's only action is to access an untrusted HTML page. The page contains a JavaScript calling navigation functions `FollowHyperlink`, `HistoryBack` and/or `WindowOpen`. The behavior of the JavaScript is modeled by a rule that conditionally appends a navigation call to the function list. As explained in Figure 9, each navigation call generates a sequence of events. It is guaranteed that all possible interleavings of event sequences are exhaustively searched, because Maude explores all viable rewrite orders.

(3) Specifying the execution context (Step (e) in Figure 2). Many Boolean conditions affect the execution path, e.g., conditions to return from a function and conditions to create a new frame. These conditions constitute the execution context of the system. We defined rules to assign both true and false values to these conditions. Therefore the `search` command explores both paths at each branch in the pseudo code. The assignments of the Boolean conditions, combined with the function call sequence, constitute a *potential* spoofing scenario. These may include false positive scenarios, in the sense that such Boolean values cannot at the same time be attained by different variables, and thus, as shown in Figure 2, mapping a potential scenario back to the real-world is important. It is a manual effort guided by the formally derived potential scenarios. We discuss this in Section 4.4.

(4) Modeling Function Calls and Events (Step (b) in Figure 2). There are three types of actions shown in Figure 9: calling a function, invoking an event handler and posting an event. A function call is implemented as a term substitution in the function call queue. For example, the function call `SetInteractive` is specified by the following rule, where `F` is the frame of *Markup* `M`, and `SetInteractive(F)` can conditionally rewrite to `SwitchMarkup(M,F)` (if `BOOLEXP1` is false) followed by `NavigationComplete(F)` (if `BOOLEXP2` is true)

Table 4: Pseudo Code and Rewrite Rule of `SetInteractive`

Pseudo Code

```
MARKUP::SetInteractive() {
  if (BOOLEXP1) return;
  this->frame->SwitchMarkup(this);
  if (BOOLEXP2)  NavigationComplete(frame)
}
```

Rewrite Rule to Specify SetInteractive

```
var F: Frame   M: Markup   FQ: FunctionQueue
rule [SetInteractive(M) ; FQ]  < M | frame: F , … >
 => [(if BOOLEXP1 ≠ true
          then SwitchMarkup(M,F) else noop fi) ;
     (if BOOLEXP2 == true
          then NavigationComplete(F) else noop fi) ;
     FQ]   < M | frame: F , … >
```

Posting of an event happens by appending the event to the event queue, for example, `FollowHyperlink` is specified by removing itself from the function queue and adding a *startNavigation* event to the end of the event queue.

```
var  U:Url F:Frame  FQ: FunctionQueue   EQ: EventQueue
rule [FollowHyperlink(U, F) ; FQ] { EQ }
 =>  [FQ] { EQ ; startNavigation(U, F) } .
```

The third type of action is the invocation of an event handler. Any event can only be invoked when its previous event handler returns. To model this restriction, any rule of an event handler invocation specifies that the first event in the event queue can be dequeued and translated into a function call only when the function queue is empty. Below is the rule to specify the handling of the `ready` event, which invokes the handler `SetInteractive`.

```
var  EQ: EventQueue
rule [empty]  { ready(M)  ;  EQ }
 => [SetInteractive(M)] { EQ }
```

5) Specifying the program invariant of address bar correctness (Step (a) in Figure 2). A good state is a state where the URL on the address bar matches the

URL of the *View* and is also the URL of the content that is painted on the screen. In addition to that, the URL is the URL of the *currentMarkup* of the *primaryFrame*. Therefore the program invariant is defined by the following `goodState` predicate:

```
vars U: URL   F: Frame    M: Markup
equation   goodState (addressBar(U) urlOfView(U)
           urlPaintedOnScreen(U) primaryFrame(F)
           < F | currentMarkup: M , ...>   < M | url: U , ...>)
       =  true .
```

It is also important to specify the initial state for the search command. In the initial state, both the event queue and the function call queue are empty. The *primaryFrame* is `f1`. The *currentMarkup* of `f1` is `m0`. The *pendingMarkup* of `f1` is uninitialized. `m0` is downloaded from `URL0`. The address bar displays `URL0`, the *View* is derived from `URL0`, and the *View* is painted on the screen. The following equation specifies `initialState`:

```
const f1:Frame  m0:Markup   url0:URL    empty:EventQueue
equation   initialState
      =  { empty } [ empty ] primaryFrame(f1)
         < f1 | currentMarkup:  m0 , pendingMarkup: nil >
         < m0 | url: url0 , frame: f1 > addressBar(url0)
         urlOfView(url0) urlPaintedOnScreen(url0) .
```

4.4 Uncovered Spoofing Scenarios

We used the `search` command to find all execution paths in the model that start with the initial state and finish in a bad state (i.e., denoted as "`not goodState`" in Maude). The search was performed on two navigations, i.e., two `FollowHyperlinks`, two `History_Backs`, one `FollowHyperlink` with one `History_Back`, and one `WindowOpen` with one `FollowHyperlink`.

Each condition shown in Table 5 is present in at least one execution context of a potential spoofing scenario uncovered by Maude. Some function names in the Location column were not shown in Figure 9, because Figure 9 only shows a sketch of the logic of navigation, while the actual model we implemented is more detailed. The `search` result in Table 5 provides a roadmap for a systematic investigation: (1) we have verified that when each of these conditions is manually set to true in the corresponding location using a debugger, the real IE executable will be forced to take an execution path leading to a stable bad state; therefore, our investigation should be focused on these conditions; (2) many other conditions present in the pseudo code are not in Table 5, such as those conditions in `SwitchMarkup`, `LoadHistory` and `CreateRenderer`, therefore these functions do not need further investigation.

The versions in our study are IE 6 and IE 7 Beta 1 through Beta 3. In the rest of this section, we will focus on conditions No. 2, 9, 11 and 18, for which we have succeeded in constructing real spoofing scenarios. For the other conditions, we have not found successful scenarios to make them real without the debugger. They may be false positives due to the fact that our model does not include the complete logic of updating and correlating these conditions, but simply assumes that each condition can be true or false at any point during the execution. In this sense, our address bar modeling is not exact (too permissive). Because of the imprecision in modeling these Boolean conditions, we need a considerable amount of effort to understand their semantics. Constructing successful scenarios is still a non-trivial "security hacking" task. Nevertheless, Table 5 provides a valuable roadmap to narrow down our investigations.

Table 5: Conditions of Potential Spoofing Scenarios

	Location	Condition
1	FireNavigationComplete	GetHTMLWinUrl() = NULL
2	**FireNavigationComplete**	**GetPFD(bstrUrl) = NULL**
3	FireNavigationComplete	ActivatedView = true
4	NavigationComplete	DontFireEvents = true
5	NavigationComplete	DocInPP = true
6	NavigationComplete	ViewWOC = true
7	NavigationComplete	ObjectTG = true
8	NavigationComplete	CreateDFU = true
9	**SetAddressBar**	**CurrentUrl = NULL**
10	SetClientSite	QIClassID()= OK
11	**LoadHistory**	**HTMLDoc = NULL**
12	CreateMarkup	NewMarkup = NULL
13	SetInteractive	pPWindowPrxy = NULL
14	SetInteractive	IsPassivating = true or IsPassivated = true
15	SetInteractive	HtmCtx() = NULL
16	SetInteractive	HtmCtx()->BindResult = OK
17	EnsureView	IsActive() = false
18	**RenderView**	**RSFC = NULL**

Scenarios based on condition 2 and condition 9 *(silent-return conditions)*. For ease of presentation, we assume there is a malicious site *http://evil* (or *https://evil*) in this section. The function call traces associated with condition 2 (i.e. `GetPFD(url)= NULL` in `FireNavigationComplete`) and condition 9 (i.e. `CurrentURL = NULL` in `SetAddressBar`) indicate similar spoofing scenarios: there are silent-return conditions along the call stack of the address bar update. If any one of these conditions is true, the address bar will remain unchanged, but the content area will be updated. Therefore, if the script first loads *paypal.com* and then loads *http://evil* to trigger such a condition, the user will see "*http://paypal.com*" on the address bar whereas the content area is from *http://evil*.

We found that both condition 2 and condition 9 can be true when the URL of the page has certain special formats. In each case, the function (i.e., `FireNavigationComplete` or `SetAddressBar`) cannot handle the special URL, but instead of asserting the failure condition, the function silently returns when the condition is encountered. For condition 9, we observed that all versions of IE are susceptible; for condition 2, only IE 7 Beta 1 is susceptible, in which case even the SSL certificate of *PayPal* is present with the faked page, because the certificate stays with the address bar. In other versions of IE, although they have exactly the same silent-returning statement, condition 2 cannot be triggered because the special URL has been modified at an earlier stage during the execution before `GetPFD` is called. However, even for these seemingly unaffected versions, having the silent-returning condition is still problematic – IE must guarantee that such a condition can never be true in order to prevent the spoofing.

These two examples demonstrate a new challenge in graphical interface design – *atomicity is important*. In the navigation scenarios, once the pending markup is switched in, the address bar update should be guaranteed to succeed. No "silent return" should be allowed. Even in a situation where atomicity is too difficult to guarantee, the browser should at least raise an exception to halt its execution rather than leave it in an inconsistent state.

***Scenario based on condition 11** (a race condition)*. Condition 11 is associated with a function call trace which indicates a situation where two frames co-exist in a *renderer* and compete to be the primary frame. Figure 10 illustrates this scenario.

The malicious script first loads *Page 1* from *https://evil*. Then it intentionally loads an error page (i.e., Page 2) in order to make conditional 11 true when `LoadHistory()` is called later. The race condition is exploited at time *t*, when two navigations start at the same time. The following event sequence results in a spoof: (1) the *renderer* starts to navigate to *https://paypal.com*. At this moment, the primary frame is `f1`; (2) the *renderer* starts to travel back in the history log. Because condition 11 is true, i.e., `HTMLDoc = NULL`, a new frame `f2` is created as the primary frame. This behavior is according to the logic of `LoadHistory()`; (3) the markup of *https://evil* in the history is switched into `f2`; (4) the address bar is updated to *https://evil*; (5) the downloading of the *paypal.com* page is completed, so its markup is switched into `f1`. Since f1 is not the primary frame anymore, it will not be rendered in the content area; (6) the address bar is updated to *https://paypal.com* despite the fact that `f1` is no longer the primary frame. When all these six events occur in such an order, the user sees *http://paypal.com* on the address bar, but the *https://evil* page in the content area. The SSL certificate is also spoofed because it gets updated with the address bar.

This race condition can be exploited on IE 6, IE 7 Beta 1 and Beta 2 with a high probability of success: in our experiments, the race condition could be exploited more than half of the time. The exploit does not succeed in every trial because event (5) and event (6) may occur before event (3) and event (4), in which case the users sees the address "*https://evil*" on the address bar.

It is worth noting that race conditions are likely to exist in the logic supporting the tab-browsing mode as well, in which multiple *renderers* share and compete for a single address bar.

***Scenario based on condition 18** (a hostile environment)*. Condition 2 and condition 9 trigger the failures of address bar updates, while condition 18 (i.e., `RSFC = NULL` in `RenderView`) triggers the failure of the content area update. We found that the condition can be true when a certain type of system resource is exhausted. A malicious script is able to create such an environment by consuming a large amount of the resource and then navigating the browser from *http://evil* to *http://paypal.com*.

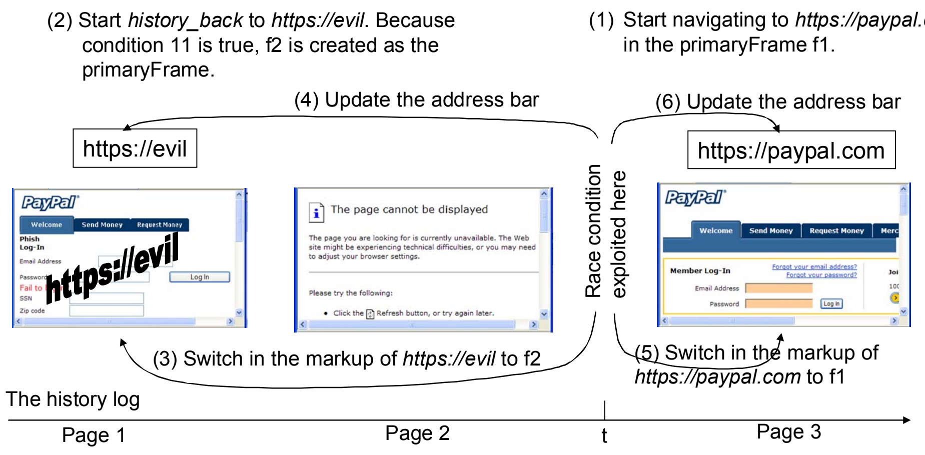

(2) Start history_back to https://evil. Because condition 11 is true, f2 is created as the primaryFrame.
(1) Start navigating to https://paypal.com in the primaryFrame f1.
(4) Update the address bar
(6) Update the address bar
https://evil
https://paypal.com
https://evil
The page cannot be displayed
Race condition exploited here
(3) Switch in the markup of https://evil to f2
(5) Switch in the markup of https://paypal.com to f1
The history log
Page 1
Page 2
t
Page 3

Figure 10: Spoofing Scenario Due to a Race Condition

When the timing of the navigation is appropriate, the browser will succeed to update the address bar and fail to update the content area, leaving the *http://evil* content and the *paypal.com* URL visible to the user.

Once again this example demonstrates the importance of atomicity in graphical interface implementations. In addition to the correctness of the internal logic of a browser, this scenario emphasizes the resilience against a hostile execution environment.

5. Discussions

In order to better put our work into perspective, this section presents higher-level discussions about possible defense techniques, other visual spoofing flaws and various techniques for GUI logic analysis.

5.1 How to Defend Against GUI Logic Exploits

The most direct defense against spoofing attacks is bug fixing. All scenarios that we have discovered have been confirmed by the IE development team. In a build after IE 7 Beta 3, all the status bar spoofing bugs and two address bar bugs have been fixed. Two other address bar bugs have been investigated, and their fixes have been proposed.

In situations where the vendor's patches are not yet available, vulnerability-driven filtering can provide fast and easy-to-deploy patch-equivalent protection. In particular, we have explored the possibility of using *BrowserShield* [18] to foil spoofing attacks. In *BrowserShield*, web pages are intercepted at a browser extension, which injects a script-rewriting library into the pages and sends them to the browser. The rewriting library is executed during page rendering at the browser, and rewrites HTML pages and any embedded scripts into safe equivalents. The equivalent safe pages contain logic for recursively applying run-time checks according to policies that detect and remove known attack patterns that we described earlier. In our proof-of-concept implementation, we authored policies for both status-bar spoofing removal and address-bar spoofing removal. The status bar policy is to inject JavaScript code into static HTML contents to monitor the status bar before the mouse click, and compare it with the URL argument of the `FollowHyperlink` call. One of the address bar policies is to inject JavaScript code to check if a URL can cause a silent failure of the address bar update.

5.2 Achieving GUI Integrity is Challenging

The objective of this paper is to bring the GUI logic problem to the attention of the research community, rather than claiming that the visual spoofing problem as a whole can be solved in the short term. In particular, the following two questions are not addressed by this work.

(1) *Is GUI-logic correctness important to users that are security-unconscious and completely ignore any security indicators?* User-studies have raised the concern that many average users still lack the knowledge or the attention to examine the information provided by security indicators, such as the address bar, the status bar, SSL certificate and security warning dialogs [6][24]. Many users readily believe the authenticity of whatever is displayed in the content area. We agree that this is the current fact, and argue that a significant effort should be spent on user education about secure browsing. But such an education would be ineffective without the trustworthiness of the security indicators – if their information can be spoofed, even we, as computer science professionals, do not know what to trust. The success of anti-phishing must be achieved by a joint effort between the browser vendors and the end users. It is analogous to automobile-safety: drivers have the responsibility to buckle up, and the automobile manufacturers need to guarantee that the seat-belts are effective.

(2) *How to deal with other types of visual spoofs that are not due to GUI logic flaws?* In the introduction, we listed a few visual spoofing scenarios due to graphical similarities. These issues have little to do with logic problems, so their treatments are very different from the approach presented in this paper. For example, the current version of IE disallows a script from the Internet zone to open a chromeless window (i.e., a window having only the content area). It is also clearly specified in design that the URL displayed on the address bar should be left-justified after each address bar update, and no pop-up window can stay "always-on-top", etc. *SpoofStick* is designed to interpret any confusing URL on the address bar [20]; *Dynamic Security Skins* [5] and *Passpet* [26] use trusted images to defeat certain spoofing attacks. Ye and Smith proposed several ideas to implement trusted paths for browsers by disallowing the page *content elements* to forge the page *status elements* [25]. Virtual machine techniques have also been used to provide trusted browser GUI elements, e.g., the *Tahoma* window manager provides a virtual screen abstraction to each browser instance [4]. Nevertheless, when the internal GUI logic is flawed as shown in the paper, ensuring unforgeable GUI elements is not a remedy. Therefore, GUI logic flaw and graphic similarity can be viewed as two different problems under the same umbrella of visual spoofing.

5.3 A Broad Spectrum of Tools Can Be Used for Systematic Exploration

The essence of our approach is that we systematically explore GUI logic. Whether the exploration is done by symbolic formal analysis (such as theorem proving or model checking) or by exhaustive testing is less important. As an example of exhaustive testing, we used the binary instrumentation tool *Detours* [10] to test the status bar logic. The basic idea is that since we know the program invariant and how to generate canonical user action sequences and canonical DOM trees, we can generate actual canonical HTML pages and actual mouse messages to test the actual IE status bar implementation. The advantage of the exhaustive testing approach is that it does not require manual modeling of the behaviors of each HTML element, and therefore can avoid the potential inaccuracies in the logic model. Applying this technique, we were able to find all spoofs derived from our previous modeling.

Nevertheless, there is no fundamental difference as to whether the exploration is done symbolically (e.g., by Maude) or by exhaustive testing (e.g., by Detours), because both techniques are based on the same understanding of the search space and the test-case construction. The main effort for the symbolic exploration is to correctly specify the GUI logic in sufficient detail. The exhaustive testing requires much effort to drive the system's internal state transitions. For example, to test the address bar logic, we would need to exhaustively enumerate all event interleaving possibilities in an actual renderer, which is a non-trivial task.

6. Related Work

The contributions of our work are: (1) the formulation of GUI logic correctness as a research problem, and (2) the proposal of a systematic approach to uncover GUI logic flaws leading to visual spoofs. There is little existing work related to our first contribution, but a wealth of work is related to the second – formal methods and program analysis techniques have been successful in discovering software reliability and security flaws. We summarize a few techniques below.

The SLAM technique [1] uses theorem proving and model checking tools to statically verify whether or not predefined "API usage rules" are obeyed in large programs. A static driver verifier is built on the SLAM technique, and has been deployed for Windows driver implementation correctness. Model checking techniques are also developed to find file system bugs [27] and security vulnerabilities [3] in large bodies of legacy source code. Much research has been done in formal verification of security protocols [15]. A static analysis technique is used for detecting higher level vulnerabilities such as SQL injections, cross-site scripting, and HTTP splitting attacks [13]. Our work is complementary to the existing research, because we have focused on machine-user link trustworthiness.

Also related are research papers on phishing attacks, e.g., *PwdHash* is a browser plug-in that transparently produces a different password for each site to prevent phishing sites from obtaining usable passwords [19]. Florencio and Herley designed a technique to detect password phishing by monitoring password-reuse patterns between a well-known site and an unfamiliar site [9].

7. Conclusions

GUI logic flaws are a real and pressing security problem – these flaws can be exploited to lure even security-conscious users to visit malicious web pages. We have formulated GUI logic correctness as a new research problem, and have proposed a systematic approach to proactively uncover logic flaws in browser GUI design/implementation that lead to spoofing attacks.

Specifically, based upon an in-depth study of the logic of key subsets of IE source code, we have developed a formal model of the browser logic and have applied formal reasoning to uncover important new spoofing scenarios. This has been done for both the status bar and the address bar. The knowledge obtained from our approach offers an in-depth understanding of potential logic flaws in the graphical interface implementation. The IE development team has confirmed that all thirteen flaws reported by us are indeed exploitable, and has fixed eleven of them in the latest build. Through this work, we demonstrate the feasibility and the benefit of applying a rigorous approach to GUI design and implementation.

Despite the fact that the analysis approach is systematic, it only provides *relative completeness*: relative to the kind of spoofing scenarios being considered, the IE code subset currently modeled, and our search spaces. Therefore, an important task ahead is to obtain a precise high-level specification of more IE modules, and to extend our current formal models and analyses to cover most IE functionality. For example, the model should accommodate the tab browsing logic and the hosting mechanisms for document types other than HTML, such as PDF, Microsoft Word, Macromedia Flash, etc. Our methodology can be extended to tackling this pending challenge in the future.

GUI logic flaws affect all web browsers, not just IE. We believe that the methodology presented in this paper can be equally applied to systematically identify vulnerabilities in other browsers. More broadly, non-browser applications, e.g., email clients and digital identity management tools [14], have similar graphical interface integrity issues. Thus, ensuring GUI logic correctness is a research direction with significant practical relevance.

Acknowledgements:

Many colleagues at Microsoft kindly provided valuable comments to improve this work. We thank Bill Bolosky, Brad Daniels, Rich Draves, Ulfar Erlingsson, Tim Harris, Cormac Herley, Emre Kiciman, Jim Larus, Madan Musuvathi, Shaz Qadeer, Kevin Schofield, Dan Simon and Chad Verbowski from Microsoft Research; Jeremy Dallman, Rob Franco, Mike Friedman, Dean Hachamovitch, Vikram Harinau, Li-Hsin Huang, Patrick Mann, Dan Plaster, Christopher Vaughan, Chris Wilson, Tong Wynn, Yin Xie, Zhenbin Xu and Geng Yang from the IE Team; Steve Adegbite, Greg Hartrell, Steve Lipner, Rebecca Norlander, Mike Reavey, David Ross, Dave Steeves and Matt Thomlinson from the Security Business Unit. Collin Jackson at Stanford University and Jun Xu at North Carolina State University offered insightful discussions. We received valuable comments from anonymous reviewers. The work of José Meseguer and Ralf Sasse has been supported in part by ONR Grant N00014-02-1-0715 and NSF Grant CNS-05-24516.

References:

[1] Thomas Ball, Sriram K. Rajamani. "The SLAM Project: Debugging System Software via Static Analysis", *ACM Principles of Programming Languages Conference*, 2002.

[2] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, at al. Maude: specification and programming in rewriting logic. Theoretical Computer Science, 285(2): 2002.

[3] Hao Chen, Drew Dean, and David Wagner. "Model checking one million lines of C code". *Network and Distributed System Security Symposium (NDSS)*, 2004.

[4] Richard S. Cox, Jacob G. Hansen, Steven D. Gribble, and Henry M. Levy: "A Safety-Oriented Platform for Web Applications," IEEE Symposium on Security and Privacy, 2006

[5] Rachna Dhamija and J. D. Tygar. "The Battle Against Phishing: Dynamic Security Skins," *Symposium on Usable Privacy and Security* (SOUPS), July 2005.

[6] Rachna Dhamija, J. D. Tygar and Marti Hearst. "Why Phishing Works". *Conference on Human Factors in Computing Systems (CHI)*, 2006.

[7] Jeremy Epstein, John McHugh, Rita Pascale, Hilarie Orman, Glenn Benson, et al, "A prototype B3 trusted X Window System," Computer Security Applications Conference, 1991.

[8] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. "Web Spoofing: An Internet Con Game," 20th National Information Systems Security Conference, 1996

[9] Dinei Florencio and Cormac Herley. "Stopping a Phishing Attack, Even when the Victims Ignore Warnings". *Microsoft Research MSR-TR-2005-142*

[10] Galen Hunt and Doug Brubacher. "Detours: Binary Interception of Win32 Functions," *Proceedings of the 3rd USENIX Windows NT Symposium*, pp. 135-143. Seattle, WA, July 1999.

[11] *Internet Explorer Window Loading Race Condition Address Bar Spoofing*. http://secunia.com/advisories/19521/

[12] José Meseguer. "Conditional Rewriting Logic as a United Model of Concurrency". *Theoretical Computer Science*, 96(1): 73-155, 1992.

[13] Benjamin Livshits, Monica S. Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis," *USENIX Security Symposium*, 2005.

[14] Microsoft Corporation. *Microsoft's Vision for an Identity Metasystem.* http://msdn.microsoft.com/

[15] Catherine Meadows. Formal Verification of Cryptographic Protocols: A Survey. Lecture Notes in Computer Science, 917, 135-150, 1995, Springer.

[16] The MSDN Library. "Changing Element Styles". http://msdn.microsoft.com/

[17] The MSDN Library. "OLE Background," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/ html/_core_ole_background.asp

[18] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. "BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML". *Operating Systems Design and Implementation*, 2006.

[19] Blake Ross, Collin Jackson, Nicholas Miyake, et al. "Stronger Password Authentication Using Browser Extensions". *Usenix Security Symposium*, 2005.

[20] SpoofStick. http://www.spoofstick.com/

[21] Firefox Visual Spoofing Flaws. Bugtraq list, http://securityfocus.com/bid. Bug IDs: 10532, 10832, 12153, 12234, 12798, 14526, 14919

[22] Internet Explorer Visual Spoofing Flaws. Bugtraq list, http://securityfocus.com/bid. Bug IDs: 3469, 10023, 10943, 11561, 11590, 11851, 11855, 1254.

[23] Netscape Navigator Visual Spoofing Flaws. Bugtraq list, http://securityfocus.com/bid. Bug IDs: 7564, 10389

[24] Min Wu, Robert C. Miller and Simson L. Garfinkel. "Do Security Toolbars Actually Prevent Phishing Attacks?" *Conference on Human Factors in Computing Systems (CHI)*, 2006.

[25] E. Ye, S.W. Smith. "Trusted Paths for Browsers." *11th Usenix Security Symposium*. August 2002.
(Also, E. Ye, Y.Yuan, S. W. Smith. "Web Spoofing Revisited: SSL and Beyond," Technical Report TR2002-417, Dartmouth College. February 2002.)

[26] Ka-Ping Yee, Kragen Sitaker. "Passpet: Convenient Password Management and Phishing Protection," *Symposium on Usable Privacy and Security*, 2006.

[27] Junfeng Yang, Paul Twohey, Dawson Engler, Madanlal, Musuvathi. "Using model checking to find serious file system errors". *USENIX Symposium on Operating Systems Design and Implementation*, 2004