A New Avenue of Attack: Event-driven System Vulnerabilities

Presented at the European Conference in Information Warfare on July 8-9, 2002, Brunel, Uxbridge, UK, *In Proceedings of European Conference on Information Warfare and Security*, MCIL, *pages 177-185*.

A New Avenue of Attack: Event-driven System Vulnerabilities

Symeon D Xenitellis

Information Security Group
Royal Holloway University of London, United Kingdom
E-mail: S.Xenitellis@rhul.ac.uk

Abstract

Hacker Warfare is the type of Information Warfare that involves the inflicting of damage to the digital infrastructure of the enemy by exploiting security vulnerabilities. In this paper we discuss for the first time the exploitation of event-driven systems in order to inflict this type of damage. As an attacker may use command line parameters and network data to exploit security vulnerabilities in local and network applications respectively, he can use events against event-driven applications.

Keywords: security, event-driven system, vulnerability, interface-based security, malware, graphical user interface

Introduction

Information Warfare has many aspects (Libicki 1995). This paper focuses on the so-called “hacker warfare”, the exploitation of security vulnerabilities in the computer systems of the enemy in order to inflict damage or disruption of services.

A software application can be regarded as an autonomous entity that can receive input from different pathways such as the network (in the case of a Web server serving requests) and command line parameters (in a privileged set-user-id application). These types of input are typically limited in number and the addition of a new one dramatically affects the process of protecting a computer system. This paper examines a new input to the software applications and in particular event-driven systems and events.

An event-driven system is typically associated with the implementation of graphical user interfaces (GUI) such as in the Windows operating systems and graphical environments, the X Window system and the Java Virtual Machine (JVM). It can also be found in Automatic Teller Machines (ATM), Point of Sale (POS) systems, certain types of smart cards, Internet kiosks and handheld devices. They either utilise the mentioned software, cut-down versions of it or other specialised software.

In an event-driven system there is typically the facility for objects to send events to other objects. Often, there is no access control for this process, even when objects belong to different users, thus it is possible for an unprivileged user to send events to objects that belong to a privileged one. The ability to send events to other objects aids an attacker in undermining the security of the system (Xenitellis 2002).

In this paper we further investigate how one can undermine the security of a system by taking advantage of the system being event-driven. We provide a high level abstraction of the types of vulnerabilities, we analyse the threat types and show how they can affect information warfare.

In Section 2 we provide a background to event-driven systems. Section 3 presents the prior work. Subsequently, Section 4 analyses the vulnerabilities and then in Section 5 we show threat scenarios that can aid in information warfare. We continue in Section 6 with countermeasures and in Section 7 we present experimental results.

Background

According to the event-driven model, an application receives sequences of events that occur asynchronously in the system (Crosbie 1996). For the GUI these could be due to user input (keyboard or mouse activity), system events (alarms or warnings) or application events (generated by other applications). An application typically responds only to a subset of those events by processing them. Since the sequence of events governs what processing takes place, the system is called event-driven system.

In the following we use the term *object* to describe any component in an event-driven system that can be a distinct receiver of events. In a GUI, an object can be a window, a dialog box or a window control that has its own identification name called *window handle* that the system can reference in order to send events. In this paper we use the term *object handle* to describe this identification name. Finally, an application in an event-driven system can be comprised of one or more objects.

In an event-driven system two basic functionalities are typically provided, the ability to enumerate (that is, to list the object handles) the available objects and the ability to send events to them.

In Table 1, the listed versions of Windows allow the objects to enumerate other objects and send events to them.

Capability	Windows 9x	Windows 2000/XP
Enumeration	Yes	Yes
Sending events	Yes	Yes

Table 1: Functionalities and implementations

Prior work

It has been shown (Forrester & Miller 2000, Miller et al. 1995) that event-driven system implementations for GUIs are bound to exhibit software reliability problems when fed with random events. In fact, Forrester (2000) has shown that all applications tested in Windows had severe reliability problems when receiving random events.

In (Xenitellis 2002) an analysis is provided on security vulnerabilities in event-driven systems. This analysis is focused on the types of impact and it compares the effect of these vulnerabilities with that of buffer overruns. On the contrary, in this paper we provide an alternative abstraction to the type of the vulnerabilities, we put focus on threat scenarios and elaborate more on the countermeasures.

Vulnerabilities

The vulnerabilities in event-driven systems can be categorised in the following basic groups. These can be vulnerabilities due to the

- richness of parameters in the events,
- sequence of the receiving of events, and
- notion of interface-based security.

Richness of semantics of parameters in events

The events in an event-driven system are composed of an event-type and event-specific data (payload). Typically the payload is several bytes of data as shown in Table 2 for the case of Win32.

Payload	Size (in bytes)
Parameter 1 (WPARAM)	2
Parameter 2 (LPARAM)	4
Total	6

Table 2: Example of payload details

The LPARAM parameter might be used to carry the address of a string or a record structure (Ezzell 1998) or even the address of a subroutine to execute. A malicious user can study the available event-types and attack an application by sending carefully crafted events.

As an example, in Windows, the WM_TIMER event type takes as a parameter an address that the program will execute upon receipt of the event (MSDN 2002). In (Forrester 2000), a similar example of such a practice is shown of a pointer being dereferenced from a parameter sent in an event.

In both cases, since events can originate from malicious sources, objects should treat the payload of events as potentially malicious as well.

In (Shelton 2000) it is shown that the Win32 API does not respond well when system calls are called with obscure parameters. Thus, such obscure data can be passed using events and compromise the security of the system.

Sequence of events being received

The objects in event-driven systems receive sequences of events and they are designed by the programmers having in mind only common sequences of events. Unorthodox sequences can influence the state of the object and possibly make it hang or crash (Forrester 2000). If these sequences are carefully selected, then they can have the potential to undermine the security of the object.

In Figure 1 we provide a naive example of an authentication application to demonstrate the issue of how a special sequence of events can bypass the security of the system. The code is flawed, however in real implementations it gives the correct results. The user has to move the mouse across a window through an invisible track without falling from it. The correct user

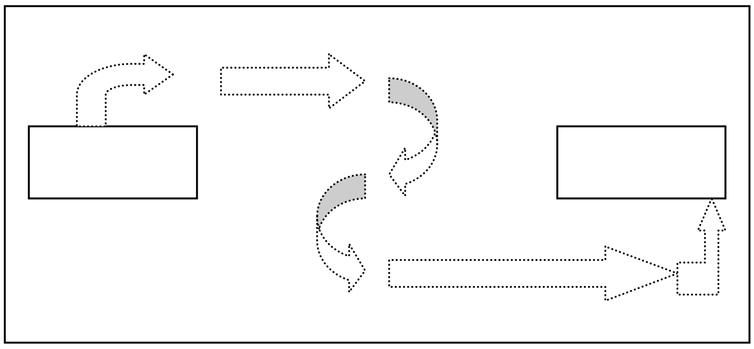


Figure 1: A naive authentication scheme

remembers what the track looks like and follows it. In the figure one can see the correct route while in the real case the route is invisible. The user has to click on the “Begin” box and drag the mouse along the route toward the “End” box where he releases the mouse. If there has been no mistake, then the user is authenticated.

If we assume that the source code looks like the following,

```
switch (event_type)
{
     case  mousebuttonPRESSED:   // the mouse button is pressed,
           if (mouse_in_begin_box())
                 dragging++;      // mouse in Begin box, we run…
           break;
     case  mouseMOVING:      // the mouse is moving,
           if (dragging > 0 && mouse_out_of_path() == TRUE)
                 dragging--; // if button pressed but made mistake,
                             // reset variable.
     case mousebuttonRELEASED:   // the mouse button is released,
           // If mouse is released in End box while we drag it,
           if (mouse_in_end_box() && dragging == TRUE)
                 authenticated = 1;    // Success, authenticated.
           else
                 dragging = 0;    // Mistake, reset variable.
           break;
}
```

The code shows that if we artificially send hundreds of mousebuttonPRESSED events to the application, without corresponding mousebuttonRELEASED events, we manage to increase the value of the “dragging” quite high. Then we can move the mouse towards the End box without fear of stepping out of the path since every mistake decreases the “dragging” variable by only one.

The error in the judgement of the programmer in this case is that each mousebuttonPRESSED event is followed by one mousebuttonRELEASED event.

Interface-based security

In the GUI it is common to put filters or restrictions for the user input in order to simplify the internal functionality of an application. With such limits, the application can expect more predictable input data and do away with complex filtering in the source code. However, due to the event functionality, it is possible to bypass these filters and restrictions and send any type of data. For example, an input box that is marked as read-only can be converted to modifiable and have its value changed.

Interface-based security is not a robust security mechanism. The same type of vulnerability is known to exist in Web sites that implement filtering of input using a scripting language such as Javascript or VBScript. Once the data in the Web page has been sanitised by the scripting language, a Common Gateway Interface (CGI) script is invoked to process them further. The CGI script assumes that the input is sane and does not perform additional checks. The attacker would invoke the CGI script directly bypassing the filtering of the scripting language.

An example of interface-based security is the facility in Windows to have a password control attribute to edit controls. If an edit control has this attribute, then it is not possible to undermine security by performing a copy and paste operation. However, if the edit control looses the password control attribute momentarily, then it is possible to perform a copy and paste operation.

Threat scenarios

We present the following threat scenarios that make use of the vulnerability types as discussed in Section 4.

Enhanced functionality in malware

Trojan horses and virii can use events to affect the functionality of antivirus software or of other protection software such as personal firewalls.

In order to terminate a victim process that is running in the background, the attacker has to enumerate the available objects and identify the correct object handle. The enumeration retrieves identifiable information of the processes. Subsequently, to close the victim process, the malware would need to send an event instructing it to terminate. The command would look like the following

```
SendMessage (<antivirus handle>, WM_CLOSE, 0, 0);
```

Additionally, the malware software could also send a known sequence of events that can crash or hang the victim. For example, the following command sends a single event that would make the victim execute arbitrary code in its address space

```
SendMessage (<antivirus handle>, WM_TIMER,
                     0, <arbitrary or known address>);
```

Assuming that we have managed to disassemble the binary code of the victim and that we have some knowledge of the names of the available subroutines, we can then call those subroutines through the above event. With this technique we not only are able to execute the code of any of the events of the victim, we can also execute any of the subroutines it contains.

Obviously, in this case of execution of any address in the address space of the victim, we do not have the ability to specify any parameters for the subroutine. If the subroutine has parameters, it pops them from the stack, eventually trashing the internal memory. However, experimentally it was found that it often manages to execute to an extent before the victim crashes.

The malware could execute code under a false identity if it could place appropriately coded machine language code in the address space of the victim. For a mailer application, this could be achieved with a binary file being sent as an attachment. This attachment could be quite big in size containing the No Operation (NOP) machine language operation. At the end of the file, there should be the exploit code that the malware wishes to execute. Using statistics, the author of the malware could estimate the address space where the attachment would reside in memory and attempt to execute at an address in that area. Statistically the malware author has a good chance of stumbling upon the NOP operations that perform no execution whatsoever and eventually reach the exploitation code. This technique for the case of buffer overruns has been described at least by Aleph One (1996).

Finally, malware could exploit event-driven system vulnerabilities in order to propagate. This can mask their signature of behaviour so that they are undetectable by antivirus programs that only check the access to the filesystems.

Intercepting passwords

Generally, in event-driven systems it is possible to intercept events sent to specific objects using the appropriate system functionality that is typically provided. However, in multi-user environments it is not possible to intercept events destined to an object that belongs to another user in order to avoid eavesdropping. If it were possible, the attacker would be able to intercept the keystrokes while a password is being typed.

However, the ability for an attacker to send events allows him to instrument a special sequence that can still capture the password (Xenitellis 2002). The attacker needs to send the appropriate events the text of the password being written and perform a copy and paste operation. But if the edit box has the password feature enabled (that is, asterisks are echoed for each key pressed), then it is not possible to copy the content. However, this protection is interface-based security and can be disabled momentarily for the copy and paste operation and enabled afterwards by sending a short series of events.

To sum up, the attacker can intercept confidential information of the victim by sending sequences of events.

Exploiting trusted devices

For a fairly complex application, it is quite difficult to ensure that it will function correctly when it receives all combinations of sequences of events. This situation is referring to the issue of maintaining a sane state for whatever sequence of events arrives at the system.

Devices such as ATMs, POS systems, certain types of smart cards and handheld devices use underlying event-driven systems. When these are used to offer a specific range of services, an attacker could either use the available input devices to insert custom sequences of events or modify the hardware to do so.

Countermeasures

The primary solution would be to re-establish, if possible, that applications execute in a single security domain. That is, each application is unaffected by other applications and cannot receive events from them. The functionality of objects sending events to other objects should be disabled. However, this would bring about two main problems, the loss in terms of usability and more importantly the need to heavily rewrite applications. Such a scenario is realistic for cut-down versions of event-driven systems such as smart cards and handheld devices where the need to communicate between applications using events is minimal. Indeed, in several of these cut-down versions of event-driven systems there is no support for inter-object communication using events.

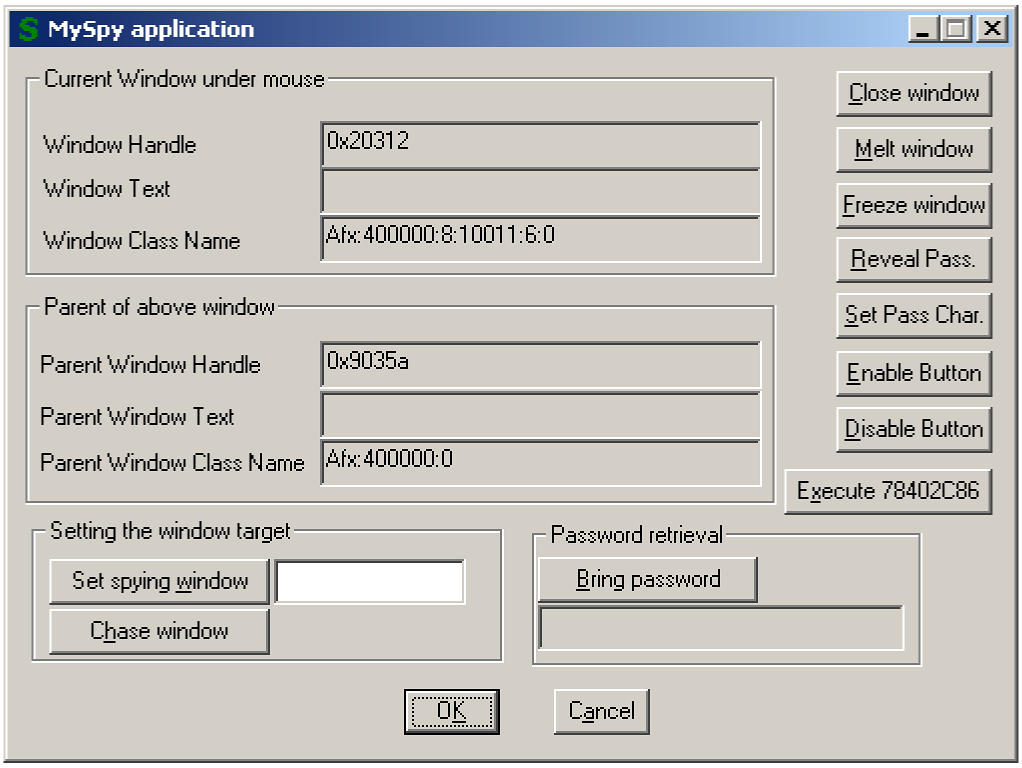
On the other hand, such systems would be still vulnerable to attacks that generate events artificially, either by modifying the input components or interfacing directly with the circuits of the device.

An event-driven application typically has an event loop that receives events and dispatches them to the appropriate internal subroutines. A level of filtering with filtering code can be put before the dispatching of the events. Alternatively, the method described by Gosh (1999) is used when it is not possible to recompile the application that we need to protect. When the application is executed, the event loop is replaced with a custom that does the filtering and passes the filtered events to the actual event loop.

Finally, currently there is no tool that can automatically detect possible unorthodox sequences of events by examining the source code and this task is typically left to the programmer. The manual checking of source code is error-prone and this inability to solve the problem efficiently has similarities with the issue of buffer overrun vulnerabilities (Xenitellis 2002).

Experimental results

These are the experimental results of the testing of the Windows operating systems and graphical environments with regards to possible sources of vulnerabilities. For demonstration purposes an auxiliary application was written to help test the vulnerabilities as shown in Figure 2.


MySpy application

Current Window under mouse

Window Handle: 0x20312

Window Text:

Window Class Name: Afx:400000:8:10011:6:0

Parent of above window

Parent Window Handle: 0x9035a

Parent Window Text:

Parent Window Class Name: Afx:400000:0

Figure 2

Dangerous event types

It was found that at least the WM_TIMER event type can be exploited to make the victim application execute on demand code that resides in the address space of the victim object. The button labeled "Execute 78402C86" sends the following event to the victim in order to execute code at a random address.

```
PostMessage(victim, WM_TIMER, (WPARAM )0, (LPARAM )0x78402C86);
```

Retrieval of sensitive information

It was demonstrated that it is possible to retrieve sensitive information without having the capability to intercept events. The only capabilities required were to enumerate objects and send events to them. This can be achieved with the "Bring password" button. The code the emits the events is

```
    PostMessage( victim, EM_SETPASSWORDCHAR, 0, 0 );
    PostMessage( victim, EM_SETSEL, 0, -1);
    PostMessage( victim, WM_COPY, 0, 0);

    PostMessage( victim, EM_SETPASSWORDCHAR, '*', 0 );
    PostMessage( victim, EM_SETSEL, 0, 0);
```

Modification of interface components

Interface-based security measures should not be relied upon as long as it is possible to send events to individual objects. It is known that read-only edit boxes can be converted to modifiable ones and allow attackers alter data that were not supposed to change. However, it was shown that even more complex objects could be manipulated. For example, buttons can be disabled or enabled such as in an authorisation screen where the correct password has to be presented in order for the *Next* button to be enabled and continue. By enabling the button programmatically, one could bypass the authorisation procedure. Again, the security of the system is not based on cryptographic techniques that require the knowledge of the password.

In Table 3 are shown the events sent for each button of Figure 3.

Button	Code fragment
Close window	SendMessage(victim, WM_CLOSE, 0, 0);
Melt window	SendMessage(victim, EM_SETREADONLY, FALSE, 0);
Freeze window	SendMessage(victim, EM_SETREADONLY, TRUE, 0)
Reveal password	PostMessage(victim, EM_SETPASSWORDCHAR, 0, 0)
Set password	PostMessage(victim, EM_SETPASSWORDCHAR, '*', 0);
Enable button	EnableWindow(victim, TRUE);
Disable button	EnableWindow(victim, FALSE)

Table 3

Conclusion

Hacker Warfare is the type of Information Warfare that seeks to damage computer systems by exploiting vulnerabilities in them. The attackers through a limited number of interfaces can access these systems while the addition of even a single new interface alters the protection mechanisms considerably.
In this paper one such new interface is presented. Computer systems that are event-driven systems can demonstrate security vulnerabilities due to the richness of the semantics of parameters in the events, the sequence of events being received and the use of interface-enforced security.

In order to evaluate better the consequences to hacker warfare, three threat scenarios to hacker warfare are presented. Malware such as Trojan horses and viruses can exploit event-driven systems in order to deliver their payload or become undetectable, confidential information such as passwords can be compromised and finally trusted devices can be attacked in a high level approach.

Subsequently, directions for countermeasures to solve event-driven system vulnerabilities are discussed. The source of the problem is that of the separation of the security domains. An event-driven system has been used as a single security domain with very limited interactions with external systems. However, currently, such a system interacting heavily with external sources (such as the Internet) and is multi-user.

Due to the introduction of a new input interface to the system, software reliability and integrity methodologies have to be adjusted while security assessments and evaluations have to be updated.

Acknowledgements

The author would like to thank the anonymous reviewers of the Journal of Information Warfare for their useful comments.

The author would like to thank Sungbaek Cho, Paolo Pagliusi and Williamson de Santos for their input to this paper.
This paper is dedicated to the memory of Paul Broome (1961-2002).

References

Aleph, O. (1996) *Smashing the stack for fun and profit*, Phrack Magazine, 7(49).

Crosbie, M. and Spafford, E. (1996) Evolving Event-Driven Systems, In *Proceedings of the First Annual Conference on Genetic Programming, pp. 273-278,* MIT Press, Stanford.

Ezzell, B. and Blaney, J. (1998) Windows 98 Developer's Handbook, electronic edition, Microsoft Developer Network (MSDN) Library.

Forrester, J.E. and Miller, B.P. (2000) An empirical study of the robustness of Windows NT applications using random testing, In *Proceedings of 4th USENIX Windows System Symposium,* USENIX.

Gosh, A. and Voas, J. (1999) Inoculating software for survivability, *Communications of the ACM (CACM)*, pp. 38-44.

Libicki, M. (1995) What is Information Warfare, Strategic Forum, http://www.ndu.edu/inss/actpubs/act003/a003cont.html, *Institute of National Strategic Studies,* No. 28.

Miller, B.P., Lee, C.P., Maganty, V., Murthy, R., Natarajan, A. and Steidl, J. (1995) "Fuzz revisited: A re-examination of the reliability of Unix utilities and services", Technical Report, Computer Sciences Department, University of Wisconsin.

MSDN (2002) WM_TIMER, *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/timers_1w6q.asp,* Microsoft Developer Network (MSDN).

Shelton, C.P., Koopman, P., and DeVale, K. (2000) Robustness Testing of the Microsoft Win32 API, In *Proceedings of International Conference on Dependable Systems and Networks (DSA2000).*

Xenitellis, S. (2002) "Security Vulnerabilities in Event-Driven Systems", *Proceedings of 17th International Conference on Information Security (IFIP/SEC2002),* Kluwer Academic Publishers.