11
SECURITY VULNERABILITIES IN EVENT-DRIVEN SYSTEMS

Simeon (simos) Xenitellis*
Information Security Group,
Royal Holloway University of London,
TW20 0EX United Kingdom
S.Xenitellis@rhul.ac.uk

Abstract The event-driven model is a model commonly used in the implementation of systems such as the Graphical User Interface (GUI). While it offers important advantages over alternative choices, it often exhibits security vulnerabilities due to its architectural characteristics in the handling of events. In this paper we examine the security vulnerabilities of event-driven systems and define the conditions that produce them. We show that a substantial number of these vulnerabilities follow the same principles with buffer overrun vulnerabilities and finally we provide countermeasures.

Keywords: software security, event-driven system threats, active attacks, trojan horse, graphical user interface security

1. INTRODUCTION

An event-driven system [Berson, 1992] is a system of objects which interact with each other using a message-passing mechanism. This mechanism is controlled by a distinct component that is usually called the *event dispatcher*, and acts as an intermediary between objects. The data communicated are called events and they can originate from input devices in an unprocessed form (*raw event*) or they can be a result of communication between objects. The objects receive events in the form of *event messages*, typically of a fixed length and made up of an event type identifier and the event parameters. Each object has a designated programming procedure called *event procedure* that invokes individual procedures called *event handlers* for each type of event message.

*The author's studies are funded by the State's Scholarship Foundation (SSF) of Greece.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35586-3_46

M. A. Ghonaimy et al. (eds.), *Security in the Information Society*

To illustrate how an event-driven system works, suppose in a GUI the user clicks the left mouse button[1] in the client area of the window of a drawing application. This generates a raw event that contains the mouse position and which mouse buttons were pressed at that moment. The event dispatcher receives the raw event and adds information such as the application it is destined for, creates an event message and places it in a queue for the recipient application to pick it up. The recipient application checks for new messages and finds it. Subsequently, the event procedure is executed and chooses the suitable event handler for the specific mouse event message.

This paper describes security vulnerabilities that can arise in environments that support the event-driven model. The source of these vulnerabilities is twofold; any object is generally able to send events to any object without restrictions and specially crafted sequences of events can easily make an object malfuction.

Although we mainly provide examples on event-driven GUI systems, such as the Microsoft Windows range of operating systems and the Java Virtual Machine (JVM) implementations, these vulnerabilities can be found in any event-driven system. The scope of this paper is to cover generic event-driven systems.

This paper is divided into six main parts. In the following section we provide a background in the event-driven systems. In section 3 we list the related work. In section 4 we describe the event-driven system vulnerabilities. In section 5 we provide an analysis and in the last two sections we list countermeasures and end with conclusions.

2. BACKGROUND

In the evolution of computer engineering, the client / server architecture emerged to replace the monolithic systems of the mainframe computing [Berson, 1992]. A type of client / server architecture is the event-driven model and it is commonly used in the design of GUIs for modern multi-tasking operating systems or in real-time applications [Lorin and Deitel, 1981, page 69]. In the latter case, it is also called *queue-driven*, in contrast with the *process-driven* model.

In the event-driven model (Figure 1), the *event dispatcher* acts as an intermediary between the input devices and the applications. The event dispatcher is the server and the applications are the clients.

The event dispatcher receives input from the input devices as events and formulates them into event messages. Normally, one application is active at a

[1] Assuming a right-handed user.

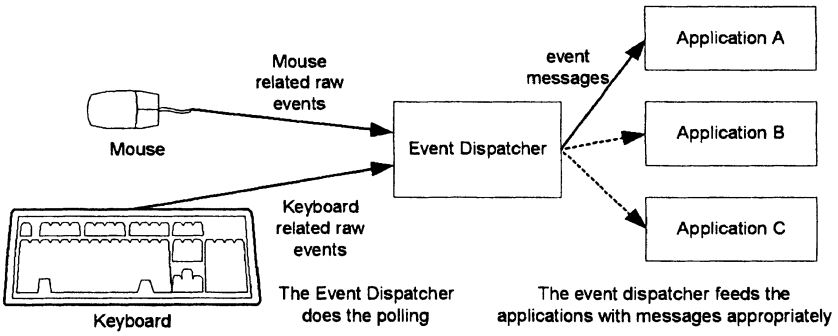

Figure 1. With the event-driven model

time. The event dispatcher knows which application that is and directs the flow of event messages to it.

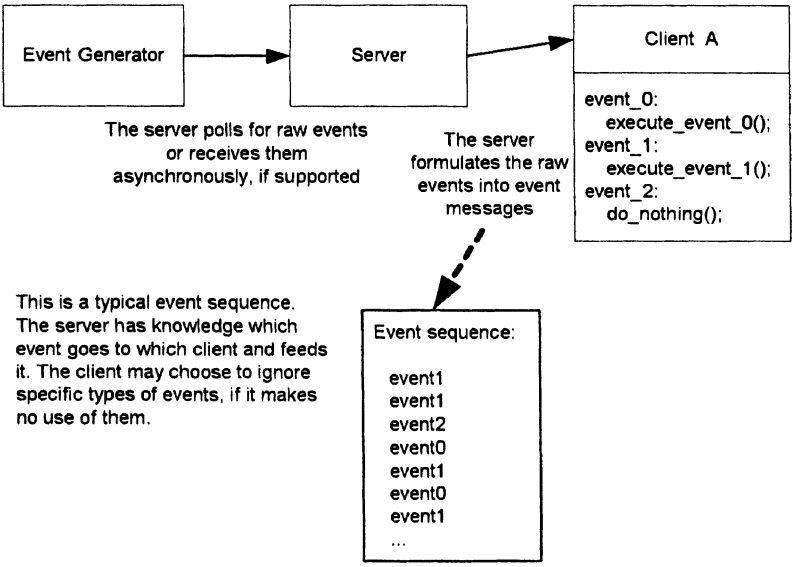

Figure 2. A generic event-driven application

A generic event-driven system (Figure 2) is comprised of the event generator that generates raw events, a server or event dispatcher that does the processing and encoding of the raw events into event messages and the client(s) that receive them for processing. The client(s), depending on its functionality, pro-

vides execution code (also known as message handlers) only for the events that are used in the specific application. As shown, the client provides execution code for events numbered 0 and 1 while event 2 is not used, thus implicitly ignored. The event dispatcher may send event messages of all three event types, however, the client will execute something interesting only for types 0 and 1. Additionally, the client is in a passive mode, one that accepts event messages from the event dispatcher, identifies the event types and forwards them to the proper execution code fragment. This puts the client in a position of dumb processing all event messages being sent. Event messages can come at any time, in any order and they can even be generated by the clients themselves, for the purposes of interprocess communication.

3. RELATED WORK

In [Forrester and Miller, 2000, Miller et al., 1995] it is shown that when event-driven applications are fed with random sequences of events, they tend to malfunction. In fact, all the applications tested in [Forrester and Miller, 2000] were found to be vulnerable to random event messages. Among the applications tested, they managed to find code fragments where the parameters of the event messages were trusted enough to be used to dereference pointers which is a very dangerous security practice. In another situation, the identification of an application that was received in a message was trusted to the degree that it was used directly without verifying the correctness of the value.

The difference between [Forrester and Miller, 2000, Miller et al., 1995] and this paper is that the former examine mainly the problems in event-driven systems as a software reliability issue. In this paper we examine them in the security perspective.

In [Ghosh and Voas, 1999, pages 38–44], the problem of software reliability with regards to commercial off-the-shell (COTS) software is examined and a method of software inoculation is presented. Using such an inoculation technique, a filtering layer is put in place that protects the operating system's system calls from invocations with invalid parameters.

4. EVENT-DRIVEN SYSTEM VULNERABILITIES

We present specific conventions to aid in the analysis of an event-driven system's vulnerabilities. Additionally, where it is appropriate, we provide examples using the GUI event-driven system.

4.1. Conventions

The purpose of the event-driven system is to manage efficiently multiple objects that are executed at the same time and interact with each other and with the environment. The victim and the attacker can be objects of the same event-

driven multitasking system. However, with the advent of distributed systems, it is possible for these two entities to be objects of different event-driven multitasking systems. As long as the distributed system supports the passing of events between distributed objects, this discussion still applies. The attacker can also be a hardware device capable of sending event messages to objects. This can be achieved by manipulating the input devices.

A custom event message is defined as an event message that may have any of its fields set to any value. Moreover, as *object enumeration* we describe the retrieval of descriptive data of the available objects in an event-driven system. These data should be sufficient to identify which applications are running. Additionally, they should contain appropriate information to allow another object to send events to it.

The victim can be an application already running or one initiated by the attacker and controlled by sending appropriate event messages.

Modern operating systems provide a set of privileges that may be given to users. Typically, administrators hold several privileges while other users have far less. Objects, in this scenario processes, that are executed under each user normally inherit their privileges. Thus one can make comparisons between processes as being more or less privileged than the other depending on whether they possess or lack a privilege. A security violation can take place if an attacking object can take advantage of the privileges owned by the victim object. This can be done by exploiting a security vulnerability.

4.2. Conditions

Different event-driven systems have different characteristics with regard to the handling of events. Since we cannot capture all systems in a single group, we devise a set of *conditions* or requirements of existence of characteristics. While describing the vulnerabilities in section 4.4, we present which conditions should be met so that they can take effect.

- **Condition Enumeration** requires that objects must be able to enumerate the objects of the event-driven system and retrieve descriptive information that could help to mount an attack.

  The enumeration of the objects is a common facility and there are generally no restrictions imposed as to which objects can be enumerated.

- **Condition Sending** requires that objects must be able to send event messages, possibly custom ones, to other objects. This includes, but is not restricted to, unprivileged objects sending events to at least one type of object of higher privilege.

  The ability to send events to any recipient without any access controls is common in event-driven systems and is typically due to their architec-

ture. This was observed when the author implemented a simple multi-threaded event-driven system. Specifically, to send an event to another object, an application invokes an exported procedure of the event dispatcher that places the event in the event queue of the system. The event dispatcher checks its queue for undispatched messages and distributes them to the individual object queues. The method of placing the message in the message queue of the event dispatcher does not allow for an efficient and foolproof mechanism to identify the source.

- **Condition Interception** requires that objects must be able to intercept event messages from other objects. This includes unprivileged objects intercepting events of objects of higher privilege.

  The interception facility is normally used for debugging purposes and to aid automated software testing.

4.3. Practical reference

As an example, we describe which conditions described in 4.2 are available in typical event-driven systems. We list the Windows9x/NT/2000 operating systems and common Java Virtual Machine implementations.

Table 1. Matrix of Conditions and Event-driven environments

Condition	*Windows 9x*	*Windows NT/2K*	*Java*	*Java Plug-in*[3]
Enumeration	Yes	Yes	No[2]	No[3]
Sending	Yes	Yes	No[2]	No[3]
Interception	Yes	No[1]	No[2]	No[3]

In Windows 9x there are no enforced access control mechanisms and consequently there is no provision for protection of the passing of events between processes. In Windows NT/2000 there is an access control mechanism and processes cannot intercept messages that are destined to a process of another user.

We provide sample code for the three conditions presented in 4.2 using the Windows operating systems as a reference.

- **Condition Enumeration**

```
EnumWindows((WNDENUMPROC )lpMyEnumFunc, (LPARAM )myarray);
```

[1] It is available when the attacker and the victim are processes belonging to the same user.
[2] It is available to applets originating from the same codebase.
[3] Because each applet is executed in a separate JVM.

This command sets up a callback function called *lpEnumFunc* and passes it to the system to execute it iteratively for each available window. This function can use the second argument to return back to the main program the identification information of the windows found.

- **Condition Sending**

```
PostMessage(hWnd_victim, WM_CLOSE, 0, 0);
```

This command sends an event message to the victim process that in this example forces it to terminate.

- **Condition Interception**

```
g_hExistingHook = SetWindowsHookEx( WH_KEYBOARD,
       (HOOKPROC )g_HookProc, g_hInstance, 0);
```

This command sets a hook, a procedure that will be able to receive all the keyboard related messages before the actual recipient.

As a side note, in Windows 2000 a user can execute an application with another identity by invoking the *runas* command-line command.

In the case of Java as it is available in browsers, Java applets are able to run in the same virtual machine instance and communicate with each other. However, this communication is quite restricted, posing minimal threat. For example, some browsers require that applets originate from the same server or additionally originate from the same directory path on the server. Newer browsers do not have internal Java support and they manage to run Java applets by making use of a plugin called the Java Plug-in. Typically, each applet of the same web page is run in a different virtual machine, making inter-applet communication using events infeasible.

4.4. Types of vulnerabilities

We give a list of types of vulnerabilities that can arise by the exploitation of events in event-driven systems. These types are classified by the security effect of the exploitation, and along with the descriptions we provide the conditions that need to be met so that each attack is possible.

4.4.1 Denial of service. Conditions Enumeration and Sending must be met in order to perform a denial of service attack. A sequence of custom event messages is sent to the victim by the attacker. The victim malfunctions, leading to loss of availability to the victim itself and possibly to the event-driven system as well.

Evidently, the victim object has to be susceptible to crashing once such a sequence of event messages is received. However, as [Forrester and Miller, 2000, Miller et al., 1995] have shown, this is quite common.

In a drawing application, the user can draw a line by holding down the mouse button, dragging the mouse and then releasing mouse button. In this three-step procedure, the drawing application receives the following messages from the mouse device.

```
WM_LBUTTONDOWN WM_MOUSEMOVE WM_MOUSEMOVE WM_MOUSEMOVE WM_LBUTTONUP
```

The attacker can send a second WM_LBUTTONDOWN before the mouse button is actually lifted up. Such a situation looks quite unlikely to happen and the programmer will probably not check the validity of the source code if this happens. If a counter (instead of a flag) was used to keep track whether the mouse button is down or up, then the application could get confused and would consider that the mouse button is pressed down all the time.

The mouse software of a popular mouse device manufacturer offers the option to simulate the double-click message using a single click of the middle mouse button. That is, to send two WM_LBUTTONDOWN events, each taking place quickly one after the other. Thus, the user can hold down the left button and click the middle button resulting in three messages of clicking down before the left button is released. Furthermore, it has been tested that in all version of Windows where this mouse software is supported (that is, from the early Windows 3.1 up to Windows 2000), the user is able to use the middle button facility to confuse the scrollbar component in any application. The user simply has to click either one of the arrow buttons and drag the mouse over the scrollbar, position the mouse on the scrollbar and finally click the middle button. The result is a visually deformed scrollbar.

4.4.2 Modification of running application. Conditions Enumeration and Sending must be met in order to modify a running application.

This modification is common practice even in legitimate applications. If the application wants to change the state of a text box from enabled to disabled so that is shows that it is no longer in use, it sends a *disable* message to the specific component. Obviously, the same mechanism can be used to change a text box from read-only to a modifiable one and vice versa.

However, apart of the technical feasibility of modifying a running application, an attacker can find other uses. For example, when installing an application, the user is typically shown an end-user license agreement (EULA) that he or she has to abide to in order to continue with the installation. This agreement is shown in a text box that is normally read-only so that it cannot be modified. In order to continue with the installation, the user has no other option than to select the *Accept* radio button, implicitly accepting the license shown. However, an attacker can easily send a crafted event message to the EULA text box that will convert it to a modifiable one. Then, it can be edited at will before choosing *Accept* and continuing. The existence of such a modification proce-

dure does not invalidate the license since the attacker intentionally modifies the license material. However, it seems to be easy for the casual attacker to believe that he or she is bypassing the license. In such situations of confusion, the vendors should disallow any possible modifications.

4.4.3 Unauthorised access to objects. When Conditions Enumeration and Sending are met, any object can send events to any other object. Thus, an unprivileged object can manipulate a privileged object by sending events.

In the Windows operating systems, an unpriviledged trojan horse can send an appropriate sequence of events to add a new administrative account, if the Administrator is logged on. Otherwise, the trojan horse can momentarily disable a personal firewall in order to communicate with the Internet.

Alternatively, the attacker can intercept the password of the administrator while the latter is trying to authenticate himself to a service. The authentication procedure generally involves the entering of a username and password in two textboxes. The password textbox has a property set that shows asterisks in the place of each character of the password typed. A further property is that it is not possible to perform a copy and paste operation on this textbox. The inability to perform a copy and paste operation is due to the first property that hides the user input. If there was no such protection, the attacker could send the messages

```
PostMessage(hWnd_victim, EM_SETSEL, 0, -1);
PostMessage(hWnd_victim, WM_COPY, 0, 0);
```

to select the password text and copy to the local clipboard.

However, the attacker can bypass this protection by disabling the password-hiding property momentarily in order to perform the copy and paste operation and subsequently enabling it again with these messages

```
PostMessage(hWnd_victim, EM_SETPASSWORDCHAR, 0, 0);
...perforn copy and paste operation as above...
PostMessage(hWnd_victim, EM_SETPASSWORDCHAR, '*', 0);
PostMessage(hWnd_victim, EM_SETSEL, 0, 0);
```

This operation takes place quickly enough so that the victim does not observe a visual change.

4.4.4 Execution of malicious code. Conditions Enumeration and Sending must be met in order to execute arbitrary or malicious code.

The event message is a structure of fixed and limited length and one would expect that it does not allow malicious code to be delivered to the victim. The attacker should put the malicious code in the address space of the victim using an indirect method such as forcing to read a file. Thus, using a custom event sequence this code can be executed.

Another scenario is that of the combination with buffer overruns. The attacker sends custom sequences of event messages so that a buffer overrun can by accomplished.

An example with the Windows operating system is that of the use of the WM_TIMER message to execute custom code. In Windows 2000, an application cannot execute code that does not reside in its address space. Thus, the attacker has to find an already available function in the victim application and execute it. The command is shown below.

```
PostMessage(hwnd_Victim, WM_TIMER, (WPARAM )0, (LPARAM )0x76f7a3);
```

It is very important to be able to provide a valid procedure address as the fourth parameter. It has been tested with the address of the PostQuitMessage() function that closes the running application and with the address of an internal function in the victim application.

4.4.5 Event Interception. Conditions Enumeration and Interception must be met in order to identify the victim object and intercept the events destined to it. Being able to intercept the events sent to an object allows the attacker to breach the confidentiality of only one direction of the object communication with the system.

In a GUI environment, intercepting the keyboard events can easily reveal authentication details that have been typed with the keyboard.

In an other situation the randomness in the key pair generation of a public key algorithm is based on the user moving the mouse in random directions on the screen. Intercepting the mouse movement can lead to the calculation of the private key.

4.4.6 An additional avenue of attack. This refers to Conditions Enumeration, Sending and Interception and the ability to attack hardware devices like smart cards or other secure event-driven devices.

In such devices, the attacker could generate artificial raw events that can subsequently be injected electronically into the event-driven device. This can cause the hardware device to malfunction, bypass security barriers or even reveal confidential information. This area is an unexplored alternative avenue to direct attacks against smart cards or similar devices.

Other hardware devices that can be victims of this attack could be *Point Of Sale* systems. These typically offer a variety of input devices like touchscreens, mice or keyboards. Again, it is possible to inject electronically custom raw events. For example, it is common in some configurations of systems that make use of the mouse to disable the right button (that is used to provide a menu of options) using mechanical alterations to the device itself. An attacker could re-enable this functionality. The right button can be used to show the

properties and valid actions of the screen components. This can be used to invoke the system shell and allow full access to the device.

In the airports of a major European city there are Internet kiosks that enable users to navigate the Internet for free and make purchases. It is possible to restart or freeze the system simply by dragging and dropping a component from the windows of the process located at the lower area of the screen to the window of the process located at the top. Subsequently, it could be possible to interfere in the booting process and take over the system. The reason the application causes the system to restart is because one of the two applications running was apparently not coded correctly when it receives the *WM_PASTE* message.

Finally, in electrically unstable environments, electrical disturbances may be translated into custom raw events and trigger malfunctions.

5. ANALYSIS

The vulnerabilities presented in 4.4 occur for two reasons.

Firstly, there are generally no access control mechanisms with regard to the enumeration of objects, the sending of events and the interception of events in event-driven systems. This is mainly due to performance degradation by adding access control and the handling complexity of the access control rules.

Secondly, it is generally difficult to control the impact of malformed events being sent to an object. An object changes state through interactions with the environment. If its security gets compromised, then it is due to its internal logic not being able to filter efficiently the input.

The inability to control efficiently the input to an object is also manifested in the buffer overrun[2] [Gollmann, 1999, Aleph One, 1996, Smith, 1997] vulnerabilities. In this type of vulnerability, the improper bounds checking of the internal buffers can cause other data structures of the victim to be overwritten with custom data, leading to a security compromise.

We show the similarities between buffer overruns and vulnerabilities in event-driven systems.

- Both of them generally surface in the implementation phase. While designing an application, the software analyst is working on a high level of abstraction that does not check array boundaries on all input of data or safeguards the event-driven objects from receiving events in a custom sequence. As shown in Section 3, these issues are left to the skills of the programmers and the aids of the language or developing environment to capture them. While in small applications it is relatively easy to avoid mistakes, in large ones the complexity appears to be overwhelming.

[2]Also know as buffer overflows.

- Both of them can make the applications to malfunction. It has been quantified in both buffer overruns [Dawson et al., 1997, Forrester and Miller, 2000, Miller et al., 1995, Shelton et al., 2000], and event-driven system vulnerabilities [Forrester and Miller, 2000, Miller et al., 1995].
- Both of them should exploit some sort of privileged applications in order to be considered that they undermine the security. In both scenarios exist applications running with different privileges. While these vulnerabilities are present in all types of applications, they appear to be considered non-critical to their vendors [Miller et al., 1995].
- Both of them give a relative measure of the quality of the software, when checked in the big ratio of unprivileged objects.
- Both of them lead to denial of service that when investigated may allow malicious code execution.

Currently, buffer overruns are more researched than the vulnerabilities in the event-driven systems.

6. COUNTERMEASURES

Any countermeasure should focus on the two weaknesses of the event-driven systems. The inability to determine the origin of an event and the difficulty to ensure that an object will function correctly for all combinations of events received.

In order for a malicious object to attack an event-driven system, it should be able to do an enumeration of the available objects and then send events to them. A mechanism in the system should restrict object enumerations and the sending of events for objects that specifically request it. That is, an object should be able to become partially invisible to other objects. The application itself should be visible to avoid hiding of trojan horse objects, however the critical components should be able be hidden. This hiding should be irrevocable and remain during the lifetime of the object.

Additionally, there should be a security review of the available types of events and those that are potentially dangerous, such as those which pass references to memory, should be flagged as such. Then, during the security inspection of the source code, these event types should be checked whether they can be exploited.

Furthermore, the event dispatcher should identify which events have been sent from objects in contrast to the events sent from other sources like input devices or the system itself. The system procedure used by objects to send messages can flag them as insecure and forward them to a firewalling component. Alternatively, the application itself could take notice of the special flag and handle these events accordingly.

Moreover, in security evaluations of computing devices, such as automated teller machinces (ATMs), there should be testing for the threat of injecting malicious events.

Finally, methods of inoculation such as those presented in [Ghosh and Voas, 1999] could provide a level of security by filtering malicious event messages. Although this degrades performance, it is a solution to COTS software.

7. CONCLUSION

We presented a set of vulnerabilities that affect event-driven systems and undermine their security. These vulnerabilities are caused by two main reasons. Firstly, it is difficult to determine the origin and the integrity of the events sent to an application, thus enabling non-restrictive sending of events that allow very powerful control of the victim object. Secondly, it is very difficult to ensure the correct behaviour of an object when it receives specially crafted events.

We categorised the vulnerabilities based on the impact they have on the victim objects. Each vulnerability requires the existence of specific primitives by the event-driven system. For each category, we listed the primitives needed. Solving the vulnerabilities in event-driven systems requires actions in two directions. Firstly, identifying the origin of events and applying access controls. Secondly, verifying that the objects manage to function properly when specially crafted events are sent.

The malfunction of applications due to the receiving of custom events has similarities with buffer overruns and follows the same principle which says that an object to be secure it should be able to handle all input in a sanitised way. Otherwise, the attacker can cause denial of service, execution of malicious code, modification of running application, unauthorised access to objects and event interception.

Furthermore, these vulnerabilities in event-driven systems can be considered as another avenue for the attacker for exploitation and another degree of complexity for the software developer to provide security.

Finally, since it is becoming more common to have multi-user event-driven systems and event-driven systems become distributed, we believe that the event-driven system vulnerabilities will start to have negative effects very soon in software security.

ACKNOWLEDGMENTS

The author wishes to thank Prof. Chris Mitchell and Dr. Keith Martin (Royal Holloway, University of London, UK), Prof. Dieter Gollman (Microsoft Research Cambridge, UK) and Robert Carolina (Tarlo Lyons Solicitors) for their input on this paper.

REFERENCES

[Aleph One, 1996] Aleph One (1996). Smashing the stack for fun and profit. *Phrack*, 7(49).

[Berson, 1992] Berson, A. (1992). *Client-server architecture*. Computer Communications. McGraw-Hill, New York.

[Dawson et al., 1997] Dawson, S., Jahanian, F., and Mitton, T. (1997). Experiments on six commercial tcp implementations using a software fault injection tool. *Software - Practice and Experience (SPE)*, 27(12):1385–1410.

[Forrester and Miller, 2000] Forrester, J. E. and Miller, B. P. (2000). An empirical study of the robustness of windows NT applications using random testing. 4th USENIX Windows Systems Symposium.

[Ghosh and Voas, 1999] Ghosh, A. K. and Voas, J. M. (1999). Inoculating software for survivability. *Communications of the ACM (CACM)*, 42(7):38–44.

[Gollmann, 1999] Gollmann, D. (1999). *Computer Security*. Worldwide Series in Computer Science. John Wiley and Sons.

[Lorin and Deitel, 1981] Lorin, H. and Deitel, H. M. (1981). *Operating Systems*. The Systems programming series. Addison-Wesley Publishing Company Inc.

[Miller et al., 1995] 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.

[Shelton et al., 2000] Shelton, C. P., Koopman, P., and DeVale, K. (2000). Robustness testing of the Microsoft Win32 API. Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000), IEEE.

[Smith, 1997] Smith, N. P. (1997). Stack smashing vulnerabilities in the unix operating system. *http://destroy.net/machines/security/nate-buffer.ps*.