With Gaming Technology towards Secure User Interfaces

Hanno Langweg
University of Bonn
Roemerstrasse 164, 53117 Bonn, Germany
langweg@informatik.uni-bonn.de

Abstract

Technology aimed at making life easier for game developers is an issue of controversy among security experts. Objections arise out of concerns of stability of a game-friendly platform. However, this kind of programming interfaces can be used to promote security as well. We use Microsoft's DirectX platform to access input and output devices directly. Thereby we enable applications to distinguish between user actions and simulated behaviour by malicious code.

With modest effort for a developer we are able to ensure authenticity and integrity of mouse and keyboard input and the display's integrity.

1. Introduction

Computer games have almost always tried to exploit a machine's resources to full extent. Ambitious graphics and quick responses to the player's commands are key factors as regards a game's appearance. In the old days of home and personal computers games were designed to be the only application running on the machine. They were given unrestrained access to display and input device resources for fast execution. That changed with the advent of multi-tasking operating environments. Applications had to share resources with other programs being run at the same time. Game developers complained about the performance penalty imposed by this administrative overhead.

On the other hand, multi-tasking operating systems taking away full control of resources from an application have been welcomed by the safety community. A misbehaving program could be terminated while working with other applications was still possible. The operating system had full control over e.g. the display, distributing access rights to the applications. No application would be guaranteed to gain access to a specific resource.

Multi-tasking makes life more complicated for developers interested in communicating directly with a user. Security sensitive applications, e.g. creating electronic signatures for documents or on-line voting, need a wilful confirmation of the user for their actions. If the display's and the input's authenticity and integrity cannot be made certain, then there is a significant lack of security. Malicious applications, e.g. Trojan horses, may simulate users' input or lure a user into actions by way of a forged visual interface. This is countered with expensive dedicated hardware or not at all.

With the intent of pleasing game developers interfaces are introduced to enable an additional direct access to resources. Consequently execution gets faster and thus a platform becomes more attractive to games. Namely the Microsoft DirectX interface is a popular choice in the personal computer domain. It has drawn criticism for its reliance upon third-party device drivers that could threaten the overall stability of the system.

Not only games benefit from granting direct hardware access. Security sensitive applications benefit, too. Control over resources that had once been used exclusively comes back and the need for dedicated hardware lessens. In the security field execution speed and impeccable visual appearance are not the primary issues. Direct access admits authenticity and integrity albeit using otherwise shared devices. User commands cannot be altered or simulated entirely by malicious programs. Program output cannot be manipulated, leading to a 'what you see is what you sign' experience in the case of creating electronic signatures.

We present how developers can use the Microsoft DirectX application programming interface to take advantage of direct hardware access for security purposes. We cover integrity and authenticity of user input and integrity of program output. Authenticity of program output, however, is outside the scope of this paper.

The paper is organized as follows. In the next section we discuss previous and related work. We then deal with input integrity and authenticity in section three and display integrity in section four. This is followed by notes on the implementation in the fifth section.

2. Previous and related work

It has been shown that applications running on the Microsoft Windows or X-Windows platform can be vulnerable to attacks by Trojan horse programs. A popular attack is to control applications remotely by simulating user actions.[1] This is done by synthesizing user input like typing on the keyboard or by sending messages to a window requesting responses.[2]

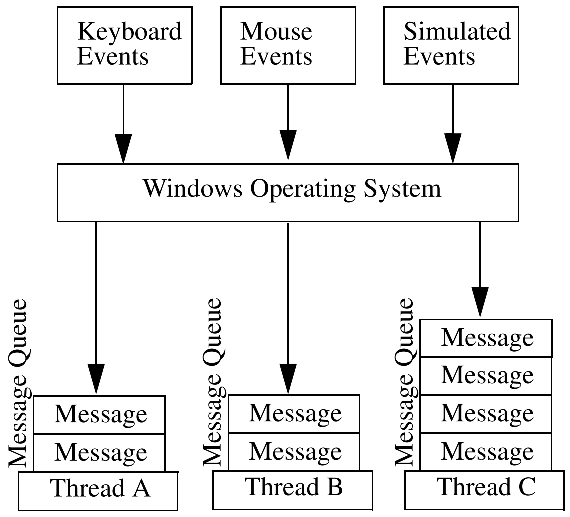
As a counter measure, messaging can be restricted at the platform or application level. The X-Windows system, for instance, allows to disable conveyance of messages placed by the *SendEvents* function.[3] In Microsoft Windows, certain types of messages can be restricted for a desktop. However, there may be occasions, like computer-based training, in which remote control of another application or parts of it is desired. Only a fraction of all applications expose an interface by which they can be explicitly automated. Consequently, simulating user input is a quick and convenient way for small helper applications.

Approaches to solve the problem of a trusted path between a security sensitive program and its user often focus on separating applications from each other. Balfanz (2001) proposes different desktops for applications to restrict interference, but this reduces co-operation as well. Janacek et. al. (2001) require a user to re-boot his personal computer in a single application mode when creating a legally binding electronic signature. Pfitzmann et. al. (2001) want to install a new operating system on users' computers, separating applications completely and allowing interaction only via a set of trusted tools.

Tygar et. al. (1996) regard window personalization as a method to make users aware whether an interface is correct or made up by a Trojan horse program. Weber (1998) argues in favour of secure personal devices that are free of Trojan horses to achieve secure user input and output.

2.1. Windows input model

Microsoft Windows uses an internal messaging model to control Windows applications. Messages are generated whenever an event occurs. For example, when a user presses a key on the keyboard and releases it or moves the mouse, a message is generated by the operating system. The message is then placed in the message queue for the appropriate thread. An application checks its message queue to retrieve messages.[4]



The system passes all input for an application to the various windows in the application. Each window has a function, called a window procedure, that the system calls whenever it has input for the window. The window procedure processes the input and returns control to the system. All aspects of a window's appearance and behaviour depend on the window procedure's response to these messages.

In the model, it is not possible to distinguish between messages placed in the queue by the operating system and messages placed by another application. To make it even worse, ordinary programs can synthesize input by help of the SendInput API function (keybd_event, mouse_event prior to NT4 SP3). This synthesized input is processed by the operating system into messages for an application. This was originally intended to assist users in operating an application by different input facilities other than the standard keyboard and mouse, e.g. assistive technology for users with disabilities. It is also a convenient tool for malicious programs.

2.2. DirectX

Microsoft DirectX is a group of technologies designed by Microsoft to make Microsoft Windows-based computers an ideal platform for running and displaying applica-

1. Cult of the Dead Cow (2002). *Back Orifice 2000.* http://bo2k.sourceforge.net
2. Spalka, A., Cremers, A.B. and Langweg, H. (2001). 'The Fairy Tale of »What You See Is What You Sign«. Trojan Horse Attacks on Software for Digital Signatures'. *Proceedings of IFIP Working Conference on Security and Control of IT in Society-II:75-86.*
3. Bråthen, R. (1998). 'Crash Course in X Windows Security'. *GridLock* 1(1998):1. http://www.hackphreak.org/gridlock/issues/issue.1/xwin.html
4. Microsoft (1998). *Microsoft Windows Architecture for Developers Training Kit.*

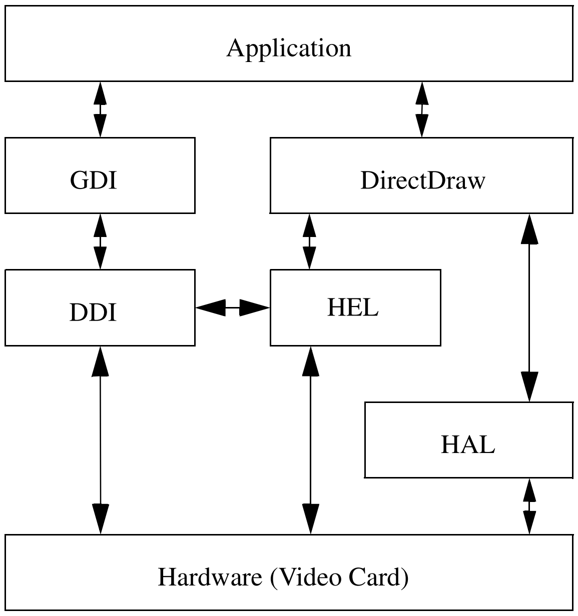
tions like games. Built directly into Windows operating systems, DirectX is an integral part of Windows 98, Windows Me, and Windows 2000/XP.

DirectX gives software developers a consistent set of APIs that provides them with improved access to hardware. These APIs control what are called “low-level functions,” including graphics memory management and rendering, and support for input devices such as joysticks, keyboards, and mice. The low-level functions are grouped into components that make up DirectX: Microsoft Direct3D, Microsoft DirectDraw, Microsoft DirectInput, to name just a few.[5] In this paper we are concerned with DirectInput and DirectDraw.

DirectInput retrieves information before it is distilled by the operating system to Windows messages. Hence, input synthesized by placing a forged message in a program’s message queue is ignored.

DirectDraw allows access to the display hardware in exclusive mode, keeping other programs from distorting the information presented to the user.

In the sketch it is shown how an application actually transfers its output to the screen. Without DirectX it uses the GDI (Graphical Device Interface) and the DDI (Display Driver Interface). With DirectX, namely its DirectDraw part, the DDI is bypassed in favour of the HAL (Hardware Abstraction Layer). If there is no direct hardware support, the HEL (Hardware Emulation Layer) is used instead.[6]



5. Microsoft (2000). ‘Microsoft DirectX Overview’. *Microsoft Developer Network Library.*
6. Microsoft (2001). ‘DirectDraw Architecture, System Integration’. *Microsoft Developer Network Library.*

3. Input integrity and authenticity

DirectInput provides an interface to get data from input devices, i.e. usually keyboard and mouse, directly before they are processed into messages for a thread’s queue. Accessing DirectInput does not require administrative privileges. From a developer’s perspective, it is less convenient compared with Windows messages. A thread has to poll a device for new input. Standard Windows controls are prepared to process messages natively; in the case of DirectInput reaction to input has to be programmed individually. Hence, it makes sense to identify security sensitive parts of an application and then replace Windows messages with DirectInput for limited sections of code.

An attacker could not only choose to forge Windows messages, but also to synthesize input by way of the SendInput API (keybd_event/mouse_event respectively). As a result key strokes and mouse movements appear as if they had been provided by a (trustworthy) device driver. This works even when using DirectInput, contradicting Microsoft’s claims that DirectInput communicates “directly” with device drivers.

To distinguish input provided by a device driver from that generated by calling SendInput one could be tempted to use a keyboard filter driver that digitally signs its data. An alternative could also be opening up a separate channel by inter-process communication means. We dismiss a separate channel because we do not want to introduce another proprietary interface that would be dependent on the driver. A filter driver, on the other hand, could attach a digital signature to a key stroke and render it using the GetMessageExtraInfo API function. The receiving application would check the signature and determine the key stroke’s validity. This poses a couple of challenges. First, the driver and the application would have to agree on a key for signed messages. Second, the scheme would have to resist replay attacks by malicious programs. Third, using the ExtraInfo value could conflict with a device driver already utilizing it for its own proprietary purposes.

We therefore take the way of prohibiting other programs from calling the SendInput function when we need undisturbed input. In consequence, forgeries can only be introduced at the level of Windows messages where they are without effect owing to DirectInput. As a side effect accessibility applications that do not run as a driver are not able to assist the user during that phase of secure input. However, this trade-off seems reasonable and inevitable here. Restricting other programs’ use of SendInput requires injecting some control code into other processes. For this, administrative privileges are required during installation of our solution.

In the fifth section we give details of our implementation.

4. Display integrity

Security sensitive information like details of a contract for which an electronic signature is to be computed should be sufficiently right at the time of use for the purpose to which the user wishes to put it. In the standard Windows GDI interface an application cannot take for granted that the area it uses to display information is unaffected by other applications. DirectDraw allows an application to acquire the display in exclusive full screen mode—games usually make use of this.

A program can create its own exclusive surface to draw on. Information on that surface cannot be altered by an attacker and thus reaches the user undisturbed. Since the surface is independent from GDI window management functions are not available and more work is put on the developer. It is advisable that it be used only for limited parts of an application where display integrity is needed. However, if developers in the security realm like to create a new user interface as much as game developers do they are free to do so. Our intention is to use DirectDraw only where it is needed to make it easy to adapt existing applications.

Utilizing the computer's display in exclusive mode is an alternative to employing dedicated secure display hardware, e.g. personal digital assistants (PDAs).

Exclusive mode does not prevent other applications from gaining foreground access. Hence, the authenticity of the display is not assured by DirectDraw methods. It may be useful to combine our approach with Tygar's et. al. window customizing (1996), making it significantly more difficult for an attacker to lure the user into false assumptions about an applications identity. Switching control away from an application can be detected and the user may then be warned.

Details of how we implement an exclusive drawing surface are given in the following section.

5. Implementation

We give an example of how to implement our approach using Windows 2000/XP. The first example shows how to retrieve key strokes, the second example presents how a form is displayed on a secure surface.

Our tests were conducted using DirectX version 8.1. However, the DirectDraw interface used is version 7. The source code is given as a Borland Delphi program. We use the DirectX header files from the Delphi-Jedi project.[7]

5.1. Capturing input from the user

In our sample we check whether the user pressed “y” to select “yes” or “n” for “no” in a confirmation dialogue.

```
SetCooperativeLevel foreground
Acquire the input device
Poll for input
GetDeviceState
Check for key pressed
```

To prohibit other (possibly malicious) programs from calling SendInput and synthesizing the user's key strokes we inject control code into running processes. This code resides in a dynamic link library (DLL) which is activated when USER32.DLL is loaded. Since SendInput is a USER32 function, loading of our DLL is assured. It is necessary to add our DLL to the key `HKEY_LOCAL_MACHINE\ Software\Microsoft\Windows NT\CurrentVersion\Windows\APPInit_DLLs` in the Windows registry.[8] This requires administrative privileges during installation.

The DLL modifies the Import Address Table of the supervised program and redirects all calls to SendInput to its own version of that function. In our test version, calls are always blocked. However, calls should be forwarded to the original SendInput function when blocking is not required. Source code of the control DLL involves several hundred lines of code and is therefore not included here.

The same effect could be achieved by employing a COTS sandboxing software for programs running on the computer that monitors and restricts the use of certain API calls including SendInput.

5.2. Drawing a window

We use the DirectDraw API to show the content of an existing form on a secure surface that cannot be manipulated. We first acquire the display in exclusive full screen mode and create the needed surfaces. Then we draw the window on the secure surface. When the secure display is no longer needed, e.g. when the user has confirmed the transaction, we stop using DirectDraw and return to standard Windows GDI mode.

The source code shows the general concept. In a commercial application a developer should bear in mind that the handling is a bit more complex. Usage would likely be

7. Delphi-Jedi Project (2002). *DirectX headers and samples.* http://www.delphi-jedi.org.
8. Microsoft (2000) ‘Working with the APPInit_DLLs Registry Value’. *Microsoft Knowledge Base Q197571.*

adaptable to the current screen resolution and colour depth. Error handling would be more extensive. An application can lose its surface when the user switches to a different program using Alt+Tab; re-acquiring a surface is necessary then.

In the first procedure we activate the usage of DirectDraw in our application and acquire the primary display in exclusive full screen mode. Two surfaces are created, the primary surface and a back surface which we use as our secure surface to draw on.

```
Create DirectDraw object
SetCooperativeLevel
   fullscreen+exclusive
Create primary surface
Create back buffer (secure surface)
```

We will now draw the content of a form to the secure surface and then flip surfaces to make the secure surface visible. The form's content is centred on the display and shown with a black background.

```
Get a device context for the
   secure surface
Draw form to secure surface
Flip surfaces
```

In the end we restore the GDI surface and allow other applications to use the computer's display again.

```
Flip to GDI surface
SetCooperativeLevel to normal
```

6. Conclusion

Trojan horse programs, i.e. programs with additional hidden, often malicious, functions, are more and more popular forms of attack. Applications that execute in an insecure environment should have control over their communication with the user.

We have presented a creative way of exploiting existing gaming technology to directly access input and output devices. Compared with dedicated hardware or operating system replacements our solution gives tractable and cost-effective means to incorporate better protection into security-sensitive programs on desktop computers.

Our work focuses on integrity and authenticity. Confidentiality of user input or program output is not discussed and remains to be scrutinized.

We are working on equipping the application developer with components to easily migrate existing programs to this secure interface. Often, only small parts of an application need heightened security. To bypass modifying programs, wrapping display integrity around a legacy application is also being examined. Incorporating input integrity may prove difficult to implement as a wrapper. To assure display authenticity we explore replacing the standard Windows shell.

7. References

[1] Balfanz, D. (2001). *Access Control for Ad-hoc Collaboration.* PhD thesis, Princeton University.

[2] Bråthen, R. (1998). 'Crash Course in X Windows Security'. *GridLock* 1(1998):1. http://www.hackphreak.org/gridlock/issues/issue.1/xwin.html

[3] CERT Coordination Center (1999). *CERT Advisory CA-99-02-Trojan-Horses.* http://www.cert.org/advisories/CA-1999-02.html

[4] Cult of the Dead Cow (2002). *Back Orifice 2000.* http://bo2k.sourceforge.net

[5] Dean, J.C., and Li, L. (2002). 'Issues in Developing Security Wrapper Technology for COTS Software Products'. *Proceedings of International Conference on COTS-based Software Systems 2002.* LNCS 2255:76-85.

[6] Delphi-Jedi Project (2002). *DirectX headers and samples.* http://www.delphi-jedi.org

[7] Fisher, J. (1995). Securing X Windows. *UCRL-MA-121788. CIAC-2316 R.0.* http://ciac.llnl.gov/ciac/documents/ciac2316.html

[8] Fraser, T., Badger, L., and Feldman, M. (1999). 'Hardening COTS Software with Generic Software Wrappers'. *1999 IEEE Symposium on Security and Privacy.*

[9] Janacek, J., and Ostertag, R. (2001). 'Problems in Practical Use of Electronic Signatures'. *Proceedings of IFIP Working Conference on Security and Control of IT in Society-II:63-74.*

[10] Lacoste, G., Pfitzmann, B., Steiner, M., and Waidner, M., ed. (2000) *SEMPER – Secure Electronic Marketplace for Europe*. Berlin et al: Springer-Verlag.

[11] Microsoft (1998). *Microsoft Windows Architecture for Developers Training Kit.*

[12] Microsoft (2002). *Microsoft Developer Network Library.*

[13] Pfitzmann, B., Riordan, J., Stüble, C., Waidner, M., and Weber, A. (2001). *The PERSEUS System Architecture.* IBM Research Report RZ 3335 (#93381) 04/09/01, IBM Research Division, Zurich. http://www-krypt.cs.uni-sb.de/~perseus/

[14] Spalka, A., Cremers, A.B. and Langweg, H. (2001). 'The Fairy Tale of »What You See Is What You Sign«. Trojan Horse Attacks on Software for Digital Signatures'. *Proceedings of IFIP Working Conference on Security and Control of IT in Society-II:75-86.*

[15] Tygar, J.D., and Whitten, A. (1996). ‘WWW Electronic Commerce and Java Trojan Horses’. Proceedings of the Second USENIX Workshop on Electronic Commerce.

[16] Weber, A. (1998). ‘See What You Sign: Secure Implementations of Digital Signatures’. *5th International Conference on Intelligence in Services and Networks, IS&N’98.* LNCS 1430:509-520.

8. Appendix: Sample code

8.1. DirectInput

In our sample we show how to check whether the user pressed “y” to select “yes” or “n” for “no” in a confirmation dialogue.

`DIK8` is the DirectInput object for the keyboard device.

```
Procedure GetUserConfirmation;
Var
   Data: TDIKeyboardState;
Begin
   DIK8.SetCooperativeLevel
   (
      Handle,
      DISCL_NONEXCLUSIVE or
      DISCL_FOREGROUND
   );

   DIK8.Acquire;
   DIK8.Poll;
   FillChar(Data,SizeOf(Data),0);
   DIK8.GetDeviceState
   (
      SizeOf(Data),
      @Data
   );
   If (Data[DIK_N] and $80) <> 0
   Then // User pressed "n"
   Else If (Data[DIK_Y] and $80) <> 0
        Then // User pressed "y"
End;
```

8.2. DirectDraw

We employ three global objects:

```
// DirectDraw object
FDirectDraw: IDirectDraw;
// primary surface
FPrimSurface: IDirectDrawSurface;
// secure surface
FSecSurface: IDirectDrawSurface;
```

In the first procedure we activate the usage of DirectDraw in our application and acquire the primary display in exclusive full screen mode. Two surfaces are created, the primary surface and a back surface which we use as our secure surface to draw on.

We tacitly assume in the sample code that a screen resolution of 640x480 suffices to display the contents of which we want to preserve integrity.

```
Procedure InitializeDirectDraw;
Var
   HR:        HRESULT;
   SurfDesc:  TDDSURFACEDESC;
   DDSCaps:   TDDSCAPS;
Begin
   FDirectDraw:=NIL;
   HR := DirectDrawCreate
         (
            NIL,
            FDirectDraw,
            NIL
         );
   If (HR = DD_OK)
   Then Begin
      // Get full screen exclusive mode
      HR := FDirectDraw.
               SetCooperativeLevel
            (
               Handle,
               DDSCL_EXCLUSIVE or
               DDSCL_FULLSCREEN
            );
      If (HR = DD_OK)
      Then Begin
         HR := FDirectDraw.
                  SetDisplayMode
               (640,480,24);
         If (HR = DD_OK)
         Then Begin
            // Create primary surface
            // and one back buffer
            // (secure surface)
            SurfDesc.dwSize:=
               SizeOf(SurfDesc);
            SurfDesc.dwFlags:=
               DDSD_CAPS or
               DDSD_BACKBUFFERCOUNT;
            SurfDesc.ddsCaps.dwCaps:=
               DDSCAPS_PRIMARYSURFACE or
               DDSCAPS_FLIP or
               DDSCAPS_COMPLEX;
```

```
          SurfDesc.
             dwBackBufferCount:=1;
          HR := FDirectDraw.
                   CreateSurface
                (
                   SurfDesc,
                   FPrimSurface,
                   NIL
                );
          If (HR = DD_OK)
          Then Begin
             DDSCaps.dwCaps:=
                DDSCAPS_BACKBUFFER;
             HR := FPrimSurface.
                GetAttachedSurface
                   (
                      DDSCaps,
                      FSecSurface
                   );
             If (HR = DD_OK)
             Then // Creation of
                  // surfaces
                  // succeeded
          End;
       End;
    End;
 End;
End;
```

We will now draw the content of a form to the secure surface and then flip surfaces to make the secure surface visible. The form's content is centred on the display and shown with a black background.

```
Procedure DrawExclusively;
Var
   DC:                 HDC;
   BkBrush:            TBrush;
   bFlippingComplete: Boolean;
   HR:                 HRESULT;
Begin
   If (FSecSurface.GetDC(DC) = DD_OK)
   Then Begin
      BkBrush:=TBrush.Create;
      BkBrush.Color:=RGB($00,$00,$00);
      BkBrush.Style:=bsSolid;
      // Create black background
      FillRect
      (
         DC,
         Rect(0,0,640,480),
         BkBrush.Handle
      );
      BkBrush.Destroy;
      // Draw form to secure surface
      fmSecure.Visible:=true;
      fmSecure.PaintTo(
         DC,
         (640-fmSecure.Width) div 2,
         (480-fmSecure.Height) div 2);
      FSecSurface.ReleaseDC(DC);
   End;
   // Flip and show secure surface
   bFlippingComplete:=false;
   Repeat
      HR:=FPrimSurface.Flip(NIL,0);
      If HR = DD_OK
      Then bFlippingComplete:=true
      Else If HR = DDERR_SURFACELOST
           Then Begin
              HR:=FPrimSurface.
                    _Restore;
              If HR <> DD_OK
              Then bFlippingComplete:=
                      true;
           End
           Else If HR =
              DDERR_WASSTILLDRAWING
           Then bFlippingComplete:=
                   true;
   Until bFlippingComplete;
End;
```

In the end we restore the GDI surface and allow other applications to use the computer's display again.

```
Procedure CleanUpDirectDraw;
Begin
   If FDirectDraw <> NIL
   Then Begin
      FDirectDraw.FlipToGDISurface;
      FDirectDraw.SetCooperativeLevel
      (
         Handle,
         DDSCL_NORMAL
      );
      If FSecSurface <> NIL
      Then FSecSurface:=NIL;

      If FPrimSurface <> NIL
      Then FPrimSurface:=NIL;
      FDirectDraw:=NIL;
   End;
End;
```