Xbox 360 Security: Best Practices

By Pete Isensee
Development Manager
Advanced Technology Group (ATG)

Published: September 29, 2006
Updated: September 2, 2010

Introduction

Xbox 360 was designed with security and anti-piracy features in mind. From anti-piracy features to built-in network authentication, the Xbox 360 console automatically handles many issues that game developers often face on other platforms. The Xbox 360 Development Kit (XDK) also provides authentication technology for protecting game content.

Not everything can be done automatically, however. Supporting a secure file system on the game disc, for example, would have significant performance implications for hashing and decrypting data. There's no reason to inflict arbitrary performance hurdles on graphical and audio bits that don't require high levels of security.

Implementing good security is part of designing and creating games. Just as developers and designers must consider how to reduce cheating and its impact on other players, they must also avoid common security errors. Much of the burden of detecting and handling game cheats falls on the shoulders of game designers and developers. This type of security is only as strong as its weakest link, and the weakest link is often the game implementation or design. This white paper describes what developers and designers can do to protect game assets, keep cheating from affecting gameplay, and maintain the integrity of the console.

Game Executables

Xbox 360 game executables are protected using public key technology. When the final version of the XEX (Xbox executable) is generated at certification time, each XEX section is cryptographically hashed using SHA-1. The resulting hash values are stored in the XEX header. The entire header (all the hash values, plus other header data) is hashed again. The resulting value is signed using the RSA algorithm. The resulting 2048-bit value, called the XEX signature, is stored in the header of the XEX file. This arrangement of sections, hashes, and signature are shown in Figure 1.

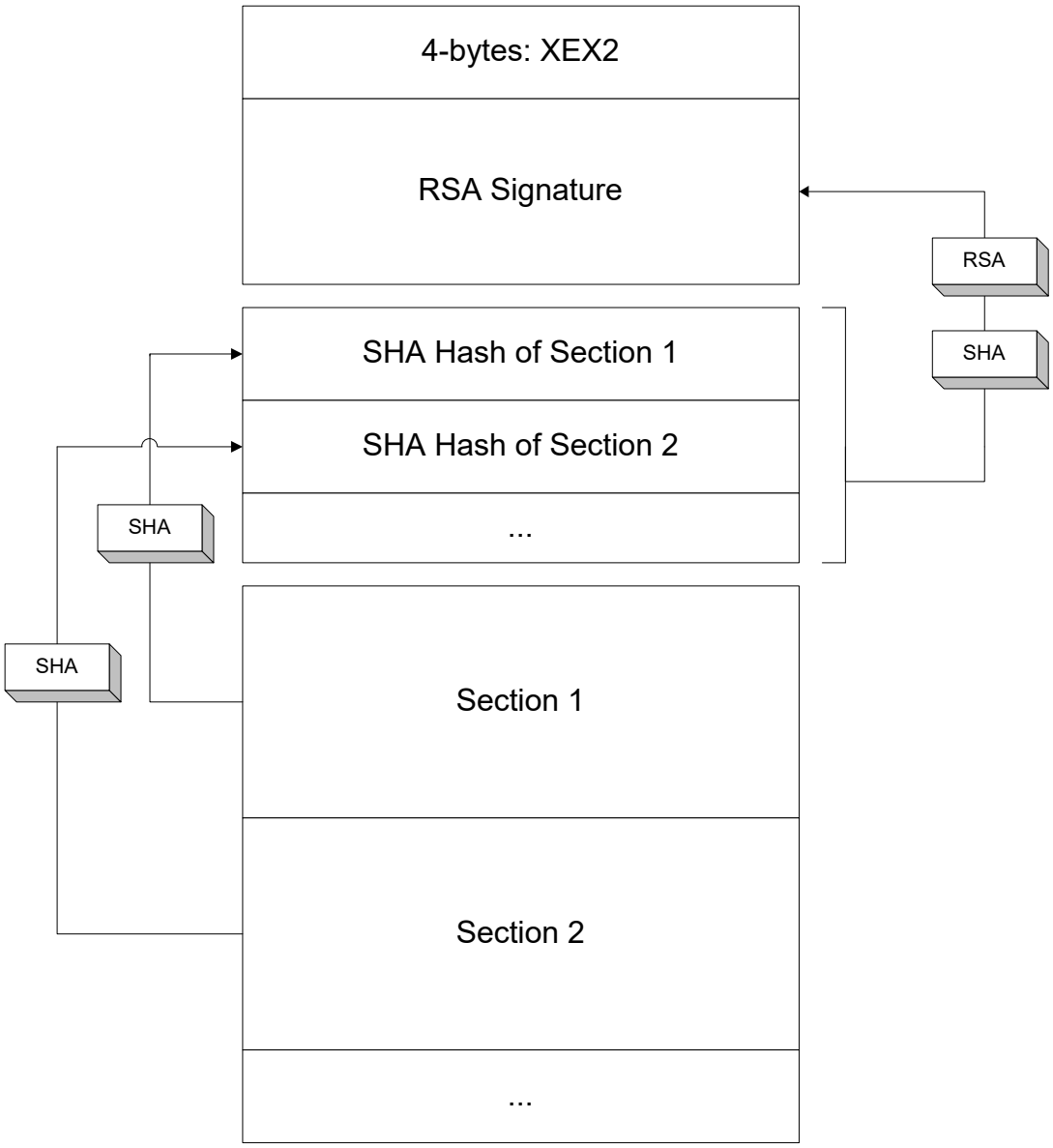

Figure 1. The Cryptographic Anatomy of a XEX File

When an Xbox 360 console loads a XEX file, it decrypts the XEX's RSA signature using the public RSA key stored in the Xbox 360 ROM. The console verifies the header by performing a SHA-1 hash of the XEX header and comparing the result with the decrypted signature. The console then verifies XEX sections as they're loaded by hashing the sections and comparing the resulting hash values with those stored in the XEX header.

To successfully crack this mechanism, a hacker must either obtain the private RSA key from the Microsoft vault, or somehow manage to factor the 2048-bit public key. The likelihood of the private key being broken by a brute force attack on the public key is infinitesimally small. You have a much better chance of being struck by lightning twice *and* winning the lottery all on the same day. Finding a modification to

a XEX section that generates the same hash requires an average of 2^{63} attempts — about three million years, assuming that you could generate a new section 100 thousand times per second.

XEX files are one of the most secure components of any Xbox 360 game. By default, XEX sections are also automatically encrypted, further obfuscating code and data from prying eyes.

Game Discs and Configuration Files

Unlike executables, general content on the Xbox 360 game disc is a ripe target for attackers. Although the DVD drive itself is authenticated, inevitably hackers will replicate game disc media and enable that media to play on Xbox 360. Although they may not be able to easily create their own executable images, it's likely they will be able to modify game content. A clever hacker could modify game disc script files, configuration information or other data to change the behavior of the game, introducing custom cheats.

To protect content, we recommend that all text files, configuration information, game parameter data, XML files, INI files, game geometry, script files, executable files (other than XEX files), and shader files be signed when they are created and authenticated whenever they are loaded. In general, any file that could be modified to give the player an advantage in the game is a likely target for hackers. Textures, audio data, video and other binary formats are generally not worth protecting in this manner. However, textures that could be modified to add transparency and allow, for example, players to see through walls, or otherwise exploit the game, should be considered for protection.

There are two primary ways to securely authenticate game configuration files:

- Store the files as resources within the game XEX image.
- Store cryptographically secure hashes of the files within the game XEX image.

Store Files as XEX Resources

Content can be embedded within the XEX itself by using the **/section** option of **ImageXEX** or **<section>** XML tag. (For additional details about ImageXEX, see the XDK documentation.) As described previously, XEX resources are automatically authenticated. Resources may optionally be encrypted. Storing configuration information in the XEX is the ultimate way to protect data.

The following example of **/section** shows how to store configuration information directly in the executable image:

```
/section:"MyConfig=config.txt,RO_ENCRYPTED"
```

In your game, load the XEX section data using the **XGetModuleSection** function:

```
VOID* pSectionData;
DWORD dwSectionSize;
if( !XGetModuleSection( GetModuleHandle( NULL ), "MyConfig",
                        &pSectionData, &dwSectionSize ) )
    return FALSE;
// pSectionData now points at the contents of config.txt
```

Store File Hashes as XEX Resources

Game configuration data can be large. In order to avoid creating huge XEX files, it is cryptographically sufficient to store secure hashes of configuration data within XEX sections. If you use this method, be sure to use a standard secure hash algorithm, such as SHA-2. Xbox 360 provides the **XHashMemory** family of functions, which support the SHA-2 family of algorithms, in addition to SHA-1 for backward compatibility. XOR checksums or CRCs are easy for hackers to regenerate by trivially changing or extending the original file, so be sure to avoid simple checksums. SHA-1 is known to have weaknesses and should only be used where necessary for compatibility. Use a secure hash utility or the Windows CryptoAPI on the configuration information during your build process, and store the resulting digest in a file. For additional security, we recommend that you also include the file size in the hash.

The following is an example of PC-side code shown (most error checking is removed for simplicity):

```
// Initialize CryptoAPI
HCRYPTPROV hProv = NULL;
if( !CryptAcquireContext( &hProv, 0, MS_ENH_RSA_AES_PROV,
PROV_RSA_AES, 0 ) )
{
    if( GetLastError() == NTE_BAD_KEYSET )
    {
        CryptAcquireContext( &hProv, 0, MS_ENH_RSA_AES_PROV,
PROV_RSA_AES, CRYPT_NEWKEYSET );
    }
}

// Generate SHA-256 digest
HCRYPTHASH hHash = NULL;
CryptCreateHash( hProv, CALG_SHA_256, 0, 0, &hHash );

// Read config.txt and hash the data
ReadFile( hFile, buffer, bufferSize, &dwBytes, NULL );
CryptHashData( hHash, buffer, dwBytes, 0 );

// Include the endian-neutral size of the file in the hash
DWORD dwSize = htonl( dwBytes );
CryptHashData( hHash, (const BYTE*)(&dwSize), sizeof(dwSize),
0 );

// Extract digest from hash context
CryptGetHashParam( hHash, HP_HASHVAL, NULL, &dwBytes, 0 );
BYTE* pDigest = (BYTE*)_alloca( dwBytes );
CryptGetHashParam( hHash, HP_HASHVAL, pDigest, &dwBytes, 0 );
```

```
CryptDestroyHash( hHash );
CryptReleaseContext( hProv, 0 );

// Write digest to file
HANDLE hDigest = CreateFile( TEXT("digest.bin"), GENERIC_WRITE,
                             0, NULL, CREATE_ALWAYS, 0, NULL );
WriteFile( hDigest, pDigest, dwBytes, &dwWritten, NULL );
CloseHandle( hDigest );
```

Then use the **/section** option to embed the digest in the executable image:

```
/section:"MyDigest=digest.bin,RO"
```

In your game, extract the digest from the XEX section using **XGetModuleSection**:

```
VOID* pSectionData;
DWORD dwStoredDigest;
XGetModuleSection( GetModuleHandle( NULL ), "MyDigest",
                   &pSectionData, &dwStoredDigest );
BYTE* pStoredDigest = (BYTE*)pSectionData;
assert( dwStoredDigest == XHASH_DIGEST_SIZE_SHA256 );
```

Whenever you read the configuration information from disc, verify that the configuration file has not been tampered with by regenerating the digest and comparing it against the one in the XEX section:

```
HRESULT hr;
XHASH_HANDLE hashHandle;

hr = XHashMemoryBegin( eXHashSHA256, &hashHandle );
if ( FAILED(hr) )
    return FALSE;

if( !ReadFile( hFile, buffer, bufferSize, &dwBytes, NULL ) )
    return FALSE;

hr = XHashMemoryUpdate( &hashHandle, buffer, dwBytes );
if ( FAILED(hr) )
{
    XHashMemoryEnd(&hashHandle, NULL, NULL);
    return FALSE;
}

DWORD dwSize = htonl( dwBytes );
hr = XHashMemoryUpdate( &hashHandle, &dwSize, sizeof(dwSize) );
if (FAILED(hr))
{
    XHashMemoryEnd(&hashHandle, NULL, NULL);
    return FALSE;
}

BYTE digest[XHASH_DIGEST_SIZE_SHA256];
hr = XHashMemoryEnd( &hashHandle, digest, sizeof(digest) );
if ( FAILED(hr) )
{
    return FALSE;
}

// See if the digests match
```

```
int comp = memcmp( digest, pDigest, XHASH_DIGEST_SIZE_SHA256 );
bool isValid = ( comp == 0 );
```

If the two digests match, then you know that the configuration file has not been modified. If the signatures are different, then either the configuration file is corrupt or it has been tampered with, and your game should not continue.

Some games have also found it effective to encrypt configuration files. Obscuring the data makes it more difficult for hackers to detect these files in the first place. However, authentication based on storing digests in the XEX is the more secure mechanism; hiding a key in the game executable is not sufficiently secure.

Hard Drive

The Xbox 360 hard drive is a primary target for attackers. On the original Xbox, game developers were required to sign data, such as saved games. However, because the signing key lived on the game disc (or console) itself, the signatures of saved games on Xbox were not very secure. Once the key was compromised, hackers could easily create and sign custom versions of saved games. By taking advantage of buffer overruns in a title's code for loading saved games, hackers were able to inject their own code into the original Xbox.

On Xbox 360, the hard drive and memory units use the Secure Transacted File System (STFS). All saved games, downloaded content, profile data (including achievements), and other content created or accessed by using the **XContent** family of functions is automatically signed and authenticated without the game developer having to do any additional work. The cryptographic system is similar to the XEX section method. Every 4K file page is signed as it's written and authenticated as it's read. Any inconsistency from tampering results in an error code, typically ERROR_DISK_CORRUPT or ERROR_FILE_CORRUPT. Unlike on the original Xbox, the validation key is safely tucked away in the Xbox 360 CPU itself. For content such as non-roaming saved games, the signing key is unique on a per-console basis.

Content stored and accessed from the 2-GB utility partition (the logical drive cache:\) of the hard disk is also automatically authenticated. This is a welcome change from Xbox, where cached content was not protected in any way. Because the utility partition is an STFS partition, the same rules of checking for ERROR_FILE_CORRUPT and ERROR_DISK_CORRUPT apply. Because STFS protects data on the hard drive in a cryptographically secure fashion, avoid adding additional corruption detection mechanisms in your game.

Memory Units

Memory units (MU) are also a significant target for attacks. On Xbox 360, all data on MUs must be created and opened using **XContent** functions, so memory unit data is automatically protected without requiring additional work. As with hard drive content, be sure to properly handle ERROR_DISK_CORRUPT and ERROR_FILE_CORRUPT error codes when reading any **XContent**-created data.

Cheating

Perhaps the most overlooked entry points for cheaters and hackers are flaws in the game design or game code. Three examples make this clear:

Pausing Games and Recording Scores

Consider any game that provides pause functionality, a common feature in games that allows players to take a break and come back to the same place. Suppose the game also records scores to Xbox Live at the end of each multiplayer session. Cheating is likely in this situation unless the designer and developer have addressed questions like the following:

- How are scores affected if one of the players in a Live session pauses the game indefinitely?
- Should the pause feature have a timeout?
- What happens if one of the players pauses specifically at key points in the game, and it prevents the other players from accomplishing their goal?
- Should there be a pause feature at all in multiplayer modes?
- Should the players have to agree to pause?

If these questions haven't been answered and implemented in the game, players could use the pause feature to artificially inflate their scores.

Causing Other Players Grief

In an early version of an original Xbox racing title, players could drive in the wrong direction on the track. Players who were losing and had no chance to catch up could simply turn their vehicle around and try to crash into other players. Although the backward racers weren't cheating in a technical sense, they had exploited a game feature in a way that the game designers had not predicted.

Exploiting Design for an Advantage

In a different racing title, there were places on the track where it was beneficial to cut through the grass to save time. In some places a player could jump over an entire patch of grass and save precious seconds. Again, although the racers weren’t cheating in a technical sense, they had exploited a game’s design in ways that the designers had not predicted. The resulting leaderboards on the grassy tracks were dominated by the players who had figured out this shortcut.

There is no way to foresee everything that a player might do in a game, and it's human nature to find clever ways of capitalizing on game features to create new and unexpected modes of fun. Many players find it fun to foil their opponents in unusual ways.

The following are best practices for finding game cheats and exploitable design:

- Make sure your beta timeframe is long enough for testers to truly exploit the game in unexpected ways.
- Assign testers to play the game with the intent to win at all costs, using any means at their disposal. For instance, removing the controller in the middle of the session or disconnecting from the network.
- It's common practice to test games by playing the so-called "sweet path." Encourage testers to stray often from the sweet path. Probe all the hidden corners of the game world. Do the things that aren't intuitive but could be allowed within the context of the game — drive backward, wear night vision goggles during the day, use the flamethrower underwater, shoot your teammates instead of the opposing team. Cheating within the framework of the game always involves the unexpected, so test the unexpected.

There will inevitably be ways to cheat in your game. Here are some best practices for reducing the impact:

- Take advantage of the Xbox Live arbitration feature (see **XSessionArbitrationRegister** in the XDK documentation). This service detects common cheats, such as removing the power or network cable mid-session, blocking network ports, or packet-flooding host machines. Using a server-based voting mechanism, Live can determine whether incoming scores are statistically valid and can throw out scores from cheaters.
- Take advantage of the Xbox Live ranking feature (see **XSESSION_CREATE_LIVE_MULTIPLAYER_RANKED**). Rankings encourage a more thoughtful and balanced game design in terms of statistics.
- Build a system into the game that penalizes cheaters. The best systems rely on social feedback mechanisms to identify cheaters. For instance, Xbox Live uses feedback to identify players who have unacceptable gamertags or abuse voice chat. Players with abnormally high negative feedback can be identified and banned from the service. See **XShowPlayerReviewUI** for details.
- One in-game method for reducing cheating is a voting system. If the majority of players decide that another player is cheating, they can vote to remove that player from the session. At the same time, keep in mind that voting systems can be exploited, too. When voting is misused, it can keep users from playing your game at all.
- Give the session host (and only the host) the authority to ban players from the session. This system allows hosts to establish any kind of game they want. If hosts want to allow the underwater flamethrower cheat, they can.
- Detect cheating when and if you can. For instance, if your game records statistics, keep an eye out for unusually large or small values. A racer recording a time of 15 seconds on a track that takes your best testers 2 minutes is very likely to have discovered a cheat.
- Don't record the scores of cheaters. Cheaters exploit games to gain personal satisfaction. Deny them the satisfaction.
- Consider subtly informing other players if you detect cheating. The best way to do this is by indicating that one player has achieved incredible powers or is otherwise exploiting the game, rather than by explicitly saying that a player

has cheated. Not only can the non-cheating players quit and move on to a better session, they can use the built-in Xbox Live feedback mechanism to report the offending player.

Mod Chips and Trainers

Many PC developers use tricks to hide important game values in memory (for example, score, health, player inventory) to protect their titles against players using trainers or other cheating software or hardware. Xbox 360 has the capability to encrypt data in RAM. For instance, executable code is loaded into encrypted memory for maximum protection. Mod chips that either tap the bus or manipulate memory won't work, because they will be changing encrypted code which will cause the game to crash. Use **XEncryptedAlloc** to allocate protected memory pages. Encrypted memory is a limited resource and should only be used to store important information like player health, player resource counts, file hash values and so forth.

Xbox 360 automatically detects modified consoles during Xbox Live sign-in. Modified consoles are not allowed online. Developers of offline games are not required to either detect or react to modified consoles. The basic philosophy about modified boxes is *Microsoft is concerned so that you don't have to be.* Microsoft has a team of engineers dedicated to researching and resolving security issues, and we will continue to improve the resiliency of the hardware and software over time.

Network Traffic and Voice

Xbox 360 network traffic is automatically encrypted and authenticated. Don't add your own encryption or checksum techniques. Xbox 360 also automatically protects against packet replay attacks, so you don't need to add your own replay detection.

Because Xbox uses secure network transmission technology, the console could be used by criminal elements for secure voice communications. To prevent this, chat data must be sent in the clear using the voice and data protocol, VDP. All other game data must be sent in encrypted form using UDP, TCP, or the game data portion of VDP. Make sure that the only traffic you send in the VDP voice data section is voice, text chat or video chat.

Although Xbox packets are quite secure from prying eyes and from modification, there are other ways to attack networked games. One way that hackers can attack an Xbox game is to block, or selectively block, packets. For instance, if an attacker can guess that packets of a certain size, or packets sent on a specific port, indicate a particular type of data, your game could be vulnerable. Prefer sending all data on a single port (port 1000). You may also consider methods for randomly adjusting packet size.

Another way that hackers can attack Xbox games is by packet-flooding their opponent. If the packet flood is large enough, it can affect the target by reducing their incoming bandwidth from other consoles or game servers. Games can do very little about this type of attack other than allow games to continue to work well even under low-bandwidth/high-latency scenarios. An attacker is more likely to focus on a

game that doesn't work well under such scenarios than on a game that has some flexibility in terms of network conditions.

Finally, hackers can attack games by unplugging their Ethernet cable at specific times or by using the standby button featured on some modems. By using these disruptions, players can pause all other players in the game.

While the Xbox network stack provides an excellent first defense against malicious attacks, there is no perfect security. Developers should not assume that network buffers are specific lengths or contain data in a certain format. Whether your data comes from the network or the hard disk, whenever your title will access external data, you should use secure coding practices and the safe C Run-Time (CRT) libraries. (Many functions in the CRT have been made more secure, and these safer versions have the suffix **_s**.)

Achievements and Gamerscore

Achievements and gamerscore are recognized as some of the most innovative features of Xbox 360. Gamers will put in an incredible amount of effort to increase their score and obtain new achievements. If there is any possible way to get achievements by cheating, you can be sure that it will be exploited.

For example, some players attempt to acquire achievements by obtaining saved games from players who have already earned the achievement. If the title doesn't properly check to determine whether the active player has truly earned an achievement, then it's easy for that player to cheat, and the achievement system loses much of its value.

To protect the achievement systems, we recommend the following:

- Use the XCONTENTFLAG_NOPROFILE_TRANSFER flag on saved games (see **XContentCreate** for details). Starting with the Fall 2006 flash update, this flag guarantees that saved games only load when the profile of the player opening the saved game is the same as the creator of the saved game.
- Consider using the CONTENTFLAG_NODEVICE_TRANSFER flag. This flag guarantees that saved games only load on the console they were initially created on.
- Before awarding achievements associated with a saved game, be sure to call **XContentGetCreator** to verify that the active player matches the profile of the creator. You don't want to award an achievement to a profile that hasn't actually earned that achievement.
- Avoid shipping with secret gamepad combos or any other methods that allow achievements to be more easily accomplished. Hidden shortcuts are never hidden for long in popular titles.
- Avoid achievements that reward a high ranking, reaching a top score on leaderboards or winning a tournament. These achievements are extremely difficult to obtain, and inspire players to figure out ways to cheat the system.
- If the game saves a player's option state to a profile, ensure that if the option state has been set to *God-mode* or *cheats enabled* that achievements are not

awarded. It's OK to have cheat codes in your game, just be sure that a player who has enabled cheat codes is not able to obtain achievements.

Xbox Live Statistics

Fortunately, developers need to do very little to maintain the integrity of Xbox Live other than use the Xbox network and Live APIs and abide by Technical Certification Requirements.

However, games that record statistics provide an open invitation to attackers and potential cheaters. As with achievements, players will go to amazing lengths to get the best score. We recommend the following tactics to maintain the integrity of the system:

- Take advantage of the Xbox Live arbitration and ranking services. These features are designed specifically to reduce the impact of cheating.
- Don't record bogus scores. A player who records 10 million frags in a two-minute session is cheating. Use thresholds to determine whether the statistics should even be recorded at all.
- Identify cheaters. Use Xbox Live leaderboards to your advantage. Record statistics that identify people who have attempted to record invalid scores. Set up matchmaking queries so that players can avoid playing with cheaters.
- Consider using hidden scoreboards to record statistics that can be analyzed to find possible cheaters, including things like distance traveled, maximum speed, and so on. Because these stats are not visible to players, it will be harder for players to detect or circumvent them.

Xbox 360 CPU

The Xbox 360 CPU contains many security features that help protect games and the integrity of the platform. For example, the Xbox 360 boot code is located in the CPU itself rather than on a ROM chip, as are console-specific cryptographic keys.

In addition, code segments on Xbox 360 are read-only and executable, but data segments (including stacks and heaps) are non-executable. This is a welcome change from the original Xbox, which had no way to flag data segments as no-execute. As a result, hackers have an additional hurdle to overcome in order to exploit buffer overruns and inject executable code.

Writing Secure Code

In comparison to many programmers, Xbox 360 developers have it pretty easy when it comes to security. For instance, using **XNet** and **XContent** functionality, network packets and files are automatically protected. However, secure coding practices are still important on Xbox 360, particularly when dealing with data coming from external sources, such as the game disc, hard drive, memory units or the network. There is no perfect security. Assume that all content that the game reads from any device, including the network and *especially* the DVD drive, is potentially bogus, and take the proper steps.

For instance, a common coding practice is to read strings from files. Consider a saved game format that has the level's name stored in the saved game file:

```
CHAR strCurrentLevel[32];
ReadFile( hFile, pSaveGame, dwSaveGameBytes, &dwBytesRead,
NULL );
strcpy( strCurrentLevel, pSaveGame );

// ...
```

This code exhibits a classic security flaw — a buffer overrun. If this saved game was successfully modified to include a level name string longer than 32 bytes, **strcpy** would happily write that string into strCurrentLevel *and beyond*, blindly overwriting the program stack. A saved file, if naively modified, would simply cause the game to crash or behave erratically when the file was loaded. However, a malicious hacker could embed custom code into the saved file that could cause the title to jump to and execute code of the hacker's choosing.

Granted, a hacker would have to overcome numerous obstacles to successfully enable a buffer overrun attack on Xbox 360. Nevertheless, the hacker community has shown time and again that they have the commitment to find and exploit flaws in games. Although it's highly unlikely that attackers could run arbitrary code, they may be able to execute arbitrary *game code*. For example, a buffer overrun could be used to call hypothetical game functions like **IncrementScore**, **MakePlayerInvisible**, or **EnableGodMode**. Buffer overruns can also be used to adjust arbitrary values in memory, such as setting damage to zero or lives to 10,000. Buffer overruns still matter on Xbox 360.

Beware of any function that accepts an unchecked buffer, like **strcpy**, **strcat**, **sprintf**, and **scanf**. The following table shows alternative secure solutions to dangerous functions. These new safe functions are available on Xbox 360 and on Windows starting with Visual Studio 2005.

Insecure Function	Secure Function
strcpy, **strncpy**	**strcpy_s**
strcat	**strcat_s**
Sprintf	**sprintf_s**
Scanf	**scanf_s**
operator >>	Restrict input size by using cin.width.

Use the compiler to help you avoid buffer overrun exploits. We highly recommend that you use #define to enable _XBOX_CRT_DEPRECATE_INSECURE in your project settings to detect dangerous functions at compilation time. Insecure functions should be replaced by their secure alternatives, which have the suffix **_s**. In many cases, switching to the safe CRT is as simple as adding **_s** to function names, because if the destination is an array, the compiler can accurately infer the array size through a bit of C++ template magic. For instance, given the code above, the only change required to make the code safe is to add **_s** to **strcpy**:

```
// Compiler "knows" that strCurrentLevel is size 32
strcpy_s( strCurrentLevel, pSaveGame );
```

If strCurrentLevel isn't an array, then an additional parameter is required:

```
CHAR* strCurrentLevel = new CHAR [32];
// Pass in size of strCurrentLevel
strcpy_s( strCurrentLevel, 32, pSaveGame );
```

If you're short on time, you can use #define to specify _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES to automatically convert many unsafe CRT calls to safe equivalents. Additional information can be found in Martyn Lovell's article in *MSDN Magazine* (May 2005), "Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries": http://msdn.microsoft.com/en-us/magazine/cc163794.aspx

Visual C++ has supported the **/GS** compiler flag for some time. This flag injects a small amount of code to detect buffer overruns. This flag is highly recommended for titles that run on Windows or on the original Xbox. However, although it's supported on Xbox 360, it adds less value there, because stack memory is not executable on the Xbox 360 console. In addition, the performance costs of **/GS** are somewhat higher on Xbox 360 than on the original Xbox and Windows.

We recommend that you avoid the use of **/GS** for performance-critical code, such as the main game loop. However, it still provides value for portions of the game that don't require high performance, such as front-end menus. It provides particularly high value for areas of the title that read external data, such as profile, downloadable content, and saved game loading code.

Many other common practices for writing secure code also apply on Xbox 360: handling integer underflow and overflow, avoiding heap overruns, and avoiding hard-coded buffers for handling any external data. Use the resources provided later in this paper, in "References," to help educate your team on platform-independent security techniques.

Xbox 360 Cryptographic Functions

The Xbox 360 console is designed so that game code generally doesn't have to apply game-specific encryption or authentication. Hard drive and MU content is automatically authenticated. Network traffic is encrypted and authenticated. However, there are some cases where using cryptographic functions is recommended or useful. For instance, protecting configuration files on the DVD requires a secure hash mechanism. Generating unique IDs for identifying a particular console can also come in handy. The XDK provides two types of cryptographic functions:

- The **XHashMemory** family of functions generates SHA-2 (SHA-256, SHA-384, or SHA-512) hashes of data. **XHashMemoryUpdate** allows developers to pass in any size of data. In addition, the update API can be called repeatedly to accrue the hash value. If this work is done on a spare hardware thread, the impact to load times is negligible.

- The **XNetRandom** function can be used to generate cryptographically secure random numbers or network *nonces*. For example, to generate a unique ID, use the following code snippet:

```
__int64 x;
XNetRandom( (BYTE*)&x, sizeof(x) );
```

Community Management

Cheating is inevitable. How your game and company is perceived, however, is very much in your control. One of the most important things you can do when it comes to cheating is to have a strong presence in the game community and gaming forums; ensure that you have a team in place to monitor and respond to issues from the people playing your title.

Community forums provide many advantages. They are a central location for players to discuss game issues and to vent their frustrations. They allow developers to hear first-hand about possible game hacks, and they provide a mechanism for giving responsive feedback. By having forums in place, you can quickly get a sense of what issues are really affecting gameplay, and you can get the help of the community in addressing those issues.

Whom Can You Trust?

All security is imperfect, but at the end of the day, you have to pick your battles and decide where to focus your coding efforts. As a game developer, you can generally trust the following data sources:

- **XEXs**. XEXs are signed with RSA keys. The private key is not on the console, nor is it ever transmitted.
- **Title Updates**. Title updates, which are also XEXs, are also signed with RSA keys.
- **XTL functions**. Data returned from Xbox Live is authenticated and guaranteed to have null-terminated strings.
- **Xbox Live strong-signed content**. Content that is *strong-signed* is signed and authenticated using Xbox Live signatures (see XCONTENTFLAG_STRONG_SIGNED), and it is quite secure. Signatures are checked using Xbox Live. Note that validating this content requires a Live connection.

You should not trust the following data sources:

- **Game disc**. Hackers have already proven that they can successfully modify game disc content (other than XEX images). Use secure signing techniques and XEX sections to protect configuration files.
- **Hard disk**. Although hard disk content is protected, you should still use safe CRT functions when accessing the data. Be sure to also handle ERROR_FILE_CORRUPT and ERROR_DISK_CORRUPT errors, and do not load invalid data.

- **MUs**. Although all MU content is protected, you should still use safe CRT functions when accessing their data. Don't forget to handle ERROR_FILE_CORRUPT and ERROR_DISK_CORRUPT errors.
- **Saved games**. Most saved games are signed with roaming signatures. It is not unreasonable that hackers may eventually determine the signing keys for at least some titles. Use safe CRT functions when accessing any external data.
- **Downloaded content**. Although downloaded content is signed, that's no reason to relax your guard. Use the safe CRT functions when accessing downloaded content.
- **Network packets**. Network packets are generally quite secure due to automatic encryption, authentication, and secure key exchange. In theory, however, a motivated hacker could emulate an Xbox by using a PC or by running unsigned code on an Xbox that emulates a game. That pseudo-Xbox title could then send packets that would correctly authenticate on other consoles, and could take advantage of a buffer overrun in the game's network code. With the forthcoming Live on Windows initiative, PCs will be on the Live network and interoperation between PCs and Xbox 360 consoles will become more and more common. It's in your best interest to use the safe CRT to process all network communication.

For untrusted data, assume the worst, and use secure coding practices to avoid buffer overruns.

Summary Recommendations

By implementing the following recommendations, you can increase the security and integrity of your game and its data:

- Use secure coding practices to avoid buffer overruns and other security holes, particularly when accessing data from the network, from the game disc, from the hard drive or memory units.
- Use the safe C Runtime (CRT) functions. Define _XBOX_CRT_DEPRECATE_INSECURE to detect insecure calls.
- Authenticate all text files, configuration information, game parameter data, XML files, INI files, game geometry and uncompiled code (shaders or scripts) when loading the data from the game disc or loading the data from the cache partition.
- Avoid creating your own cryptographic functions.
- Use standard algorithms for secure authentication, such as SHA-2. On Xbox 360, prefer the **XHashMemory** family of signing functions.
- Use cryptographically secure random number generation for network nonces. On Xbox 360, use **XNetRandom**.
- Detect and handle ERROR_DISK_CORRUPT and ERROR_FILE_CORRUPT errors when reading from the hard drive or memory unit.
- Dedicate time during the testing phase to find and fix common game cheats.
- Assume that there will be ways for players to cheat in your game, and build systems that mitigate the effects.

- Don't award achievements unless the player has actually earned them. Use the XCONTENTFLAG_NOPROFILE_TRANSFER flag to ensure that saved games are not used to circumvent the achievement system.
- Don't encrypt or authenticate network packets. The Xbox 360 socket layer does this automatically.
- Don't authenticate files opened using **XContent** routines. The XContent layer does this automatically.
- Use Live arbitration and ranking services.
- Avoid the **/GS** flag on Xbox 360 for *performance-critical* code. However, use **/GS** for code whose performance is not critical, particularly in areas of the title that read external data, such as content-loading code. We also recommend that you use the flag when building games that run on Windows.

References

The following list includes some useful references. Note that inclusion does not imply endorsement by Microsoft:

- Dudlak, Jon. "Sons of the Glitch." *1UP*, June 15, 2005. http://www.1up.com/do/feature?pager.offset=1&cId=3141457
- Hargreaves, Shawn. "MotoGP Online: An Xbox Live Launch Title Developed in Seven Weeks." *Gamasutra*, July 10, 2003. http://www.gamasutra.com/view/feature/2831/motogp_online_an_xbox_live_launch_.php
- Howard, Michael and David LeBlanc. *Writing Secure Code, Second Edition*. Redmond, WA: Microsoft Press, 2002.
- Howard, Michael and Keith Brown. "Defend Your Code with Top Ten Security Tips Every Developer Must Know." *MSDN Magazine*, September 2002. http://msdn.microsoft.com/en-us/magazine/cc188938.aspx
- Kirmse, Andrew and Chris Kirmse. "Security in Online Games." *Game Developer*, July 1997. http://www.gamasutra.com/view/feature/3212/security_in_online_games.php
- Lovell, Martyn. "Safe! Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries." *MSDN Magazine*, May 2005. http://msdn.microsoft.com/en-us/magazine/cc163794.aspx
- Pritchard, Matt. "How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It." *Gamasutra*, July 24, 2000. http://www.gamasutra.com/view/feature/3149/how_to_hurt_the_hackers_the_scoop_.php