Secure Game Design: Fundamental Concepts

By Jake Young, Senior Security Engineer
Microsoft Studios, Online Services
Published: May 8, 2012

Executive Summary

Microsoft advocates the use of the Security Development Lifecycle (SDL) to enhance security in the products that are produced and published for their platforms.

This paper provides general guidance to developers who want to make their products more tamper-resistant. This document is not to be used in lieu of SDL processes and documentation, nor does it supersede those processes and documentation.

This advice is intended to be mostly agnostic in context of the platform, protocol, and implementation details, and focuses on methods and concepts instead of a specific technology.

Contents

Secure Design vs. Security Controls

Several features differentiate Secure Design and Security Controls. Encryption, for example, is a security control. Avoiding storage of data which should be encrypted is a secure design. Secure design is much more powerful than security controls. A more secure design leads to fewer vulnerabilities that could compromise a product or service, which means the product is able to stand up longer against attacks. Secure design cannot be overcome; security controls can be. Secure design is the essence of software security.

Example of secure design in contrast with controls: A product has a requirement to allow password creation and must validate the generated password for access. The product doesn't necessarily care what the password is, just that it is the same every time for the user. There are two options for handling this case.

A Security Controls Example

Store the password, but encrypt it/decrypt it so the program can validate the password by comparing the stored password with what the user typed.

This solution relies on the strength of the encryption method used and the strength of the key management strategy chosen, in order to secure the sensitive data. The security of the data is only as strong as the security of the key.

A Secure Design Example

The user inputs a password, but this solution doesn't store it. Instead, a salted one-way-hashing routine is run (for example, SHA-2) on the password, which returns a unique hash value for the password string. The hash value can now be stored. When the user types in a password, rather than having to use cycles for encryption and key management as in Solution 1, the program computes the hash. The hash is then compared to the stored hash; if they match, the password is correct.

In this example, cryptography is used, but rather than choosing to store sensitive password data, which requires ongoing protection, the product stores a one-way hash value that could not be used to derive the user's password.

The second example is the core of secure design. The developer never had to figure out how to secure the sensitive data or the key to decrypt it because the product never stored the data to begin with – yet this achieves the same functionality as the Security Controls example.

Secure Design Principles

Avoid Storing Sensitive Data

Evaluation questions to ask:

- Will the data being stored only be used to validate something later?
- Could this data be misused/abused by an attacker?
- Could this data be valuable to an attacker?
- Could this data be collected from another trusted party and not have to persist in the product?

If the answer to any of these questions is *yes*:

- Consider salted cryptographic digests (hashes) instead of regular data.

- Obtain data from a trusted partner, such as XBOX LIVE, as required and do not store it locally.
- Transfer the responsibility of storing this data to trusted platform level offerings.
    - Data Protection API – DPAPI on Windows
    - NData (for example, NSDataWritingFileProtectionComplete)/Keychain Service on OS X/iOS
    - Keychain on Android

It should not be necessary to store the data directly with these alternatives, but if it is, the data should be stored securely.

Never Trust the Client

In any online experience, a secure design requires that the server or service never trust the client. Any client could be running under an attacker's control, and thus any data coming from a client is untrusted data . There is no trusted platform for clients; eventually every platform would be compromised. There are various exploits available online for a number of platforms, and they are not limited to any single OS, hardware configuration, or console.

If the client can be under a malicious user's control, then it must be assumed that any data generated is suspect and must be validated. This is true for both client-to-server interactions and client-to-client (or P2P) interactions.

Client to Server

This scenario is an easier scenario than client-to-client to design securely, as it is acceptable to consider a server under the control of a secure service (such as developer-owned servers, Xbox LIVE, or Application Marketplaces) as an authoritative source. However in this scenario people will often try to alter their clients to gain an edge. The following are some attributes of games that are often targeted.

Location Tampering

Speed hacking and teleporting through walls and floors are examples of an attacker attempting to tamper with their location. Often in games there is little that can be done to prevent attempts to do this kind of tampering, but it can often be detected.

- Timing Checks – In five seconds, did the player just travel 10,000 units of distance? Is that impossible? If so, it is advisable to reject that data and re-sync the client with the last known coordinates from the server.

- Sanity Checks – Is it impossible for a player to go from coordinates X,Y to X_3,Y_3 without crossing through X_2, Y_2?.
- Set Traps – Consider placing objects of interest on the map in unreachable and out of sight locations. Trigger a location-tampering rule for attempts to interact with the trap.
- Perform Logging – Periodically require the client to send its current location to the server and give its current X,Y,Z coordinates on a map. Consider logging location data into query-friendly database format so that it can be analyzed to look for patterns indicative of tampering.

Object Interaction and Currency

Objects are often a target of tampering. These could be mobile objects such as creatures and enemies or items like loot and gear.

- Atomic Commits – force an atomic commit between the client and server.
    - Client requests to interact with an object in a specific way by sending the request to the server.
    - The server receives that request and validates the interaction.
    - Server Logs the interaction and sends a 'success' message back to the client.
    - Client completes the interaction.
- Set Traps – Consider placing un-rendered objects of interest on the map and trigger a tampering rule for any attempts to interact with the un-rendered object.
- Perform Logging – Consider logging object interaction activity data into query friendly database format so that it can be analyzed to look for patterns indicative of tampering.

Note: Atomic commit protocols can scale to multiple participants but they are a resource-blocking protocol by nature so you will want to ensure the resource block is unique per player.

Consumables

Consumables such as ammunition or energy are often targets of tampering. Often consumables change quickly. Using atomic commits between the client and the server may require too much overhead for a good player experience.

Some validation points to consider with consumables in different types of games are:

- Lazy Commits – Upload extra bits of data routinely, either as a unique data stream or piggy-backed on other atomic commits. Process this data every so often when there is spare computation time, and then re-sync the client.

Example: a client has 100 units of ammo. The client begins wildly firing and kills one *target*. The target is an object and therefore requires an atomic commit to prevent tampering. Ammo data could be included as part of the atomic commit request, such as *ammo used since last request=14*. This number can decrement a counter on the server side. Periodically the server sends a message to the client syncing local ammo to the server side total.

- Sanity Checking – Watch out for behavior that doesn't make sense. Is the client consuming way too much? Way too little? These conditions can be detected.

Client-to-Client

This scenario is very difficult to validate because there is no inherent trust of either client, nor should the clients necessarily trust the data they get from one another. There are no perfect design patterns for prevention of tampering in this scenario, so in addition to the recommendations in the Data Validation section of this document, there should be heavy focus on reducing impact and scope of damage.

Some recommendations specific to client-to-client interactions follow.

Blocking Clients

Blocking is the most straightforward. Consider allowing one client to refuse interaction with another. If Client A has a negative experience with Client B, then Client A will be able to reject any attempt to instantiate a multiplayer scenario with Client B.

Limiting Competitive Interactions between Clients

A user will have the most incentive to attempt tampering with the game to gain an advantage in head-to-head scenarios. This is the worst scenario from a game design standpoint, as one player is ruining another player's experience, or even multiple players' experiences, simultaneously. If client-to-client interactions are limited to cooperative modes (including scoring), then this greatly mitigates the issue.

Reputation Tracking and Management

Allow clients to provide feedback on interactions with other players – use this feedback to increase or decrease a player's reputation score. In this scenario a player would be able to set a *reputation* threshold for themselves. This would block interactions with other players below the specified reputation score.

The problem with this system is that it can be easily abused. Organized gaming groups could choose to down-rank a specific individual, or people could use the system as a punitive one to

try to punish a player that is especially good at a competitive game. Because of these two forms of abuse this solution has to be well thought-out.

Preventing abuse of this system during design time is similar to bounds checking:

- Reputation Velocity – Maximum amount of reputation score movement in N amount of time.
- Reputation Decay – Older votes are given less weight or rolled off altogether.
- Authorization – Ensuring that only people who have had a legitimate interaction with a player can influence that player's reputation.

Multiplayer Quorum

In scenarios where there are in-game client-to-client interactions with multiple clients, or a client acting as both a server and a client, another good control can be to allow a certain number of clients to flag a specific player for removal from the game. This then brings the decision to everyone currently involved in the game for a quorum vote. If quorum is reached, the player is removed. If it is not, the match continues. This method, much like the last, can be abused.

Some considerations for designing a multiplayer quorum:

- Number of client complaints required to bring up a quorum vote.
- Minimum number of players to achieve a quorum, for example, a player asking a *friend* to join a two-player instance so they can remove the other player.
- Minimum amount of time before a client is allowed to participate in a quorum.
- Log data centrally around quorum votes for investigation and data analysis.
- If the quorum fails (player is not removed):
    - Time limit on having another quorum vote against that targeted player.
    - Time limit on the originating client's ability to *complain* and drive an additional quorum vote either against that client or any additional clients.

Save File Data

Data in save files should be carefully considered, as save files are controlled almost entirely by the user while they sit locally on a system or in cloud storage that the user is able to access. The impact of tampering with a save file could be as minimal as unlocking achievements, or much more severe in terms of player experience. Take, for example, a racing game that allows people to customize their cars. If the gear ratio is stored in the save file, attackers can find this, tamper with it, and use it to give themselves impossible gear ratios, essentially turning their cars into rockets. This ruins the multiplayer experience.

Rather than storing gameplay – changing data directly in a save file – the data could be stored as a static signed array of all allowable gear ratios, with a variable in the save file that will ultimately resolve to the allowed gear-ratio customized by the player.

Save files will be tampered with any time a user is able to access them outside of the confines of the application whether on disk or cloud storage.

Some other save file design recommendations:

- Use strong data validation routines – see the Data validation section in this document.
- Store references, not data – where possible minimize the amount of data stored in a save file and just store values that reference durable in-game data.
- Deny human readability – do not use a save-file format that allows for humans to easily read it and understand what each field does. Obfuscate the data but assume that obfuscation will eventually be broken.
- Minimize field size and number of fields – the more data an attacker has to work with, the more potential security risk there is, and the harder the data is to validate.
- Minimize complexity of data types – use simple data types where possible. It is much easier to perform strong validation against a small set of numbers than validating the contents of a string.

The vast majority of software issues that allow secured devices and platforms (for example, consoles, hand-held devices, phones, and so forth.) to be *jail broken* are caused by applications that run on devices that do not include proper validation. For game consoles and handhelds these security flaws are *almost always found in save files and the code that parses them*.

Security flaws that threaten the platform usually cause the company responsible for the platform to respond in ways that protect the platform at the expense of the game maker. Examples of these expenses include: removing the game from marketplaces, disallowing the game to be executed (for example, killbit) or forcing retail outlets to suspend sales until the issue can be resolved.

Methods and Controls to Prevent Tampering

There are several methods and controls to assist with making products more resistant to tampering or to being able to detect that tampering has occurred and enable response accordingly.

Data Validation

Nearly all software security flaws are due to poor data validation. If an application is consuming data from an untrusted source, some work must be done to validate data before attempting to write it to memory or consume it.

Instrument the code that reads data, validates the data, and writes that data to memory (commonly known as a 'parser') in a way that can be easily tested. One of the most effective ways to test data validation routines is Fuzz Testing, but if code is not instrumented in such a way that it can be isolated or extracted from the main program source and instrumented with a test harness, it will be challenging and expensive to validate.

An additional benefit of keeping parsers decoupled from other game logic is that the parsers can be reused across game titles, minimizing development time and security testing.

Trusted and Untrusted Data

Trusted Data

Trusted data can be cryptographically validated, thus verified to not have been tampered with, and which originates from a trusted party. Some examples of this:

- Data in local files that have a valid digital signature from a trusted publisher.
- The trust chain of the publisher's certificate chains to a trusted Certificate Authority.
- Cryptographically signed messages from a trusted third party, after the party's identity has been fully validated and the data can be verified as originating from that party.
- Data that contains a time stamp or token number (nonce, for example) that prevents the data from being captured and replayed.

Untrusted Data

Any data that doesn't meet the above criteria is untrusted data. The majority of all security incidents deal with not validating untrusted data.

The term untrusted data refers to data that could have been tampered with in a number of ways, at a number of times:

At Rest

- Save files, including encrypted save files.
- Data blob stored locally that doesn't have a valid digital signature from a trusted certificate authority.
- Game settings files.

- User profiles.

In Transit

- Data sent outside of an encrypted channel.
- Data that did not originate from an authenticated trusted source.
- Data sent by way of any protocol that does not use a mechanism to prevent tampering and replay attacks.
- Data sent inside of a encrypted channel (TLS1.1+/SSL) where sufficient authentication was not performed, such as:
  - Validating that the certificate has not been revoked by way of the certificate revocation list.
  - Validating that the certificate has the expected thumbprint.
  - Validation of the certificate common name and validating the certificate chains through a trusted root certificate authority.

All Microsoft game development kits, including the Xbox 360 Development Kit (XDK), XNA, XBOX Live on Windows (XBLW) and Games for Windows – LIVE (GFWL) perform the above validations when connecting back to Microsoft managed services.

Reflected Untrusted Data

Several services that Microsoft offers do not perform any data validation, as the services transport or route data between clients. An example of this is the Asynchronous Multiplayer service from XBLW. This service does no data validation, but routes the data that one client sends it to another client. While XBLW uses a trusted transport path, the data itself should be considered suspect.

Size Checking

Length of data should always be checked, and the length of fields should be minimized.

If a data field is expected to be four bytes, ensure it's exactly four bytes long, otherwise reject the data. When dealing with variable length data, determine the minimum and maximum expected size of the data, then validate that the data received does not fall outside those limits.

Field length should be minimized to reduce the impact of any potential exploits or issues. An attacker can do a lot more with an eight kilobyte field (8192 bytes) than they can with an eight byte field. It is also much more complex to validate an eight kilobyte field than an eight byte field.

Note: When doing size-checking on data that's been transformed from a multi-byte-character-set to Unicode or the other way around, never assume the ratio is static, for example, 2:1. Avoid doing these transforms where possible but if they must be done, be sure the validation code is robust.

Type Checking

Use strongly typed data and ensure all constants and variables that are defined are described with a data type. Avoid implicit type conversions; if conversion of the data to a different type is required, then convert the data explicitly, for example, Cast.

This minimizes or negates the chances of software misinterpreting maliciously modified data values and behaving in a way that was unintended, potentially causing a security issue.

Validate that stored data falls into the expected data types before consuming it. For example, the data is only ASCII, or an integer, or a floating point.

Bounds Checking

Performing bounds checking is a method to protect the product against tampering. Always consider what the maximum and minimum data bounds should be and their allowed values. Here are some examples of bounds checking:

- Gold: Value >= 0, Value <= 1,000,000,000
- Ammo: Value >= 0, Value <= 100
- Gear Ratio: Value >=5:1, Value <=.25:1
- Equipment Slots: Value >= 5, Value <= 100
- Car 12 Engine: Value >= 3, Value <= 4

This methodology should be used for any data stored in a file under an attacker's control as well as for data in transit whenever possible.

Integrity Controls

Integrity is defined as ensuring that data hasn't changed unexpectedly.

Static Files

Static files, or files shipped with a game that should not change can be validated with a number of methods.

Digital Signatures

A digital signature allows you to validate that the binary has not changed since the digital signature was applied. If the binary were to be modified then the digital signature would become invalid.

Several platforms allow for signing Portable Executable (PE) files and doing automatic validation to ensure that the signature on a PE file is valid. However, this automatic validation does not occur for non-PE files or dynamic-link library (DLL) files, which use the extension .dll. If digital signatures are being used on DLLs, they must be verified in code before loading the DLLs into memory. Consider pushing data into resource DLLs if they are in a format that cannot be easily signed.

Digital Catalog files

Microsoft Windows also supports digital catalog files. Catalog (.cat) files were created because digital signatures did not cover non-PE files and developers required the ability to ensure that non-PE files had not been tampered with. A Catalog file contains a cryptographic hash that corresponds to a file. This cryptographic hash can be used to validate the integrity of the file it corresponds to. The Catalog files' integrity is protected by way of a digital signature.

Validation of catalog files must be called in code, but it is a straightforward way to check the integrity of non-PE files.

AppX

AppX, the new Windows Store file container, has integrity checks built in for all the files that are in the container. AppX provides integrity checks for files installed on disk from the AppX as well as ensuring that the files stored in the AppX container itself are valid. If these files are tampered with, the entire Windows Store application fails and the user is prompted to re-download and re-install the application.

AppX provides this integrity check by using an XML file stored in the digitally signed AppX container that contains cryptographic hashes of all the files stored on disk.

"Hardstyle"

If, due to lack of platform support, you cannot protect resources through digital signatures or catalogs consider this alternative as a last resort:

1. Generate an asymmetric RSA-2048 key pair.
2. Hash the resource file using a secure hashing algorithm (for example, SHA2).
3. Encrypt the hash from step 2 with the RSA Private Key.

4. Code the encrypted hashed value (from step 3), along with the public key (step 1) into the validation routines.
5. On load, hash the resource value.
6. Decrypt the value coded in the binary with step 4.
7. Validate the hashes match.

Note: This method is only to be used as an option of *last resort* and is effectively creating your own cryptographic system. We strongly recommend the engagement of a security professional for code review.

Dynamic Files

Dynamic files, or *save* files, are created or modified on the client system. Dynamically generated files are among those that are very hard to secure.

The Xbox 360 console allows a save file to be signed with a roaming signature or a console signature. The roaming signature is intended to provide integrity validation and allow the save file to be moved between consoles. Due to advancements in technology, the roaming signature can now be defeated. The console can also sign a save file with a strong console signature; this signature is much harder to defeat. The drawback of using this method is that the save file will only be able to be accessed by the console that signed it. The impact to customers should be well-understood for each product using this implementation.

Other platforms may not have the ability to provide strong signatures to dynamically created files. In these cases, implementing encryption with strong integrity protocols is a method that can be used to provide tamper resistance to dynamic files.

Network Streams

Tampering with data *on the wire* can be easily prevented by using the correct secure network transport protocols with secure methods to exchange keys.

The primary way network streams are attacked is by way of a *man in the middle* attack (MITM). MITM attacks are defined as when an attacker sets up a device that intercepts (or proxies) the traffic on the wire to tamper with it before sending that traffic back down the wire.

Data should be transferred by way of a protocol that uses:

- Secure Message Authentication Codes (MACs) to ensure data integrity.
- Public Key Infrastructure (PKI) for key management.
- Microsoft development kit frameworks (XDK, XNA, GFWL or XBLW) when interfacing with Microsoft managed services.

Known good protocols:

- Transport Layer Security (TLS) 1.1+
- Xbox Secure Protocol (XSP)
- Microsoft XHTTPS

Encryption and Keys

The concept of encryption cannot be discussed without first discussing key management. No matter how strong the cryptography, if the keys are poorly managed or easily accessible, then no amount of encryption is going to protect application or game data.

Key Types

There are two classes of keys that are commonly used:

- Symmetric keys
  A symmetric key uses the same key to both encrypt and decrypt the payload. So if the key is Password1234, then the encryption protocol will use that key to both encrypt and decrypt game data.

  The problem with Symmetric keys is once the key is known by an attacker it can be shared publically, which compromises the confidentiality of the data encrypted with the key.

- Asymmetric keys
  These keys work in pairs. There is a public key, and a private key. What one key does, the other can undo. If encryption is done with the private key, the public can decrypt, and the other way around. The purpose of this scheme is to allow entities to keep their private keys secure and make their public keys available to anyone. Asymmetric keys are most commonly used to secure network traffic and data streams.

  For example, if the goal was to send encrypted data to another party and be certain that only that party could decrypt it, then the data should be encrypted with the party's public key. The remote party would then be able to decrypt the data using their private key.

  While Asymmetric keys are very secure, they tend to be slower than symmetric keys, and the private keys must be managed and stored securely to prevent an attacker from obtaining both keys in the set.

Key Management

Many game developers hard-code a pre-generated symmetric encryption key into the game binary itself. This is not advised as the key will become widely known once one person discovers it.

An excellent key type is a *statically derived key*, which is a key that can be derived given the same data points. For example, the client can look up a static GUID on the system it is installed on and use it along with another static data point and run a set of mathematical computations or a hashing algorithm to derive a key. If this key is used to encrypt and decrypt a file, then the program requires the ability to consistently reproduce the key. If the key cannot be reproduced the encrypted data will be unrecoverable.

Encryption Agility

Encryption should be instrumented in a way that allows the cryptographic algorithm and key generation methods to be easily changed. Eventually the methods a product uses to generate and store keys or the cryptographic algorithm *will be* broken. When this happens, programs must be quickly updated and the key, key management system, or cryptographic algorithm must be replaced.

Supported Algorithms

Consult the SDL regarding which cryptographic protocols are currently the most secure. The older a protocol gets, the more likely it is to have had flaws discovered or have been compromised and be considered less secure.

Recommended Algorithms

We do recommend these cryptographic algorithms that are known to be more secure at the time of this writing:

- **Symmetric Block:** AES with 128-bit key or larger
- **Block Cipher Modes**: CBC, CTS with proper initialization vector that is a random number
- **Symmetric Stream:** None – Block Cipher is much stronger, but RC4 with a key length equal or greater than 128 bits is minimally acceptable
- **Asymmetric:** RSA with a key length equal to or larger than 2048 bits, Diffie-Hellman (DHKE) with a key length equal to or larger than 2048 bits, Elliptic Curve Cryptography P-256 or greater
- **Hashing:** SHA-2 (includes: SHA-256, SHA-384, SHA-512)

Less Effective Algorithms

We do not recommend the use of these cryptographic algorithms that are currently known to be less secure:

- **Symmetric Block:** DES, 2 Key 3DES, DESX, RC2, Skipjack
- **Block Cipher Modes:** ECB, CYLINK_MEK
- **Symmetric Streams:** SEAL, RC4 with less than a 128-bit key
- **Asymmetric:** RSA with less than a 2048-bit key, DHKE, with less than a 2048-bit key, 1024-bit DSA
- **Hashing:** SHA-0, SHA-1, MD2, MD4, MD5

Note: Encryption does not guarantee integrity. Always use integrity methods and protocols.

Summary

Secure design is the essence of software security. Writing secure entertainment software is critical not only to the success of the individual software titles but also to the success of the platform. Don't store sensitive data. Never trust the client. Always thoroughly validate untrusted data. Use integrity controls where possible and encryption controls when necessary.