Digital Rights Management (DRM) System: License Generation and Implementation

2. License Generator

The license is generated as a binary file containing the following elements:

1. PC ID: 32 bytes

2. Key: 16 bytes

3. Generation Time: sizeof(time\_t) bytes (architecture-dependent, either 32 or 64 bits)

4. RSA Digital Signature: 64 bytes

It's important to note that although these sizes are used in the current version, they may change in future versions.

3. License Components

3.1 PC ID

The PC ID serves as a unique identifier for the client machine[3]. It is generated using a combination of hardware characteristics to ensure that the license is bound to a specific device.

3.2 Key

The key is the cornerstone of the license, used for encrypting and decrypting protected data. Various approaches exist for key generation, with an ideal algorithm producing unique keys without collisions[4]. However, implementing such an ideal system would require maintaining an extensive database and constant validation to prevent key repetition. This approach presents challenges:

1. It demands significant server resources.

2. It limits the total number of possible licenses to 8^(key size).

3. An ideal key generator that is entirely unpredictable is theoretically impossible to achieve.

In practice, an ideal key generator is unattainable. Our approach to minimizing collisions involves using the PC ID and generation time as seeds for a pseudorandom function[5]. This method operates under two assumptions:

1. Each user has a unique PC ID.

2. The pseudorandom function behaves similarly to a true random function.

Under these conditions, the probability of key collisions is approximately 1 in 2^256, which we consider sufficient for most applications.

For pseudorandom number generation, we use the RAND\_seed and RAND\_bytes functions from the OpenSSL library[6]. This approach offers an additional advantage: it potentially allows for key recovery in case of issues without compromising the key itself. While this feature is not implemented in the current version, it remains a consideration for future enhancements.

3.3 Generation Time

The generation time is recorded to assist in reducing key collisions and can be used to implement license expiration features. This value is obtained using the time(NULL) function from the ctime library.

3.4 RSA Digital Signature

The RSA digital signature serves multiple purposes in the license[7]:

1. Signing the License:
   * When creating a license file, the provider calculates a hash (a fixed-size representation) of the license data.
   * This hash is then encrypted using the provider's private key, creating the signature.
   * The signature is attached to the license file.

Verification:

* + When the software needs to verify the license, it first calculates the hash of the license data.
  + It then uses the provider's public key to decrypt the signature, revealing the original hash.
  + If the calculated hash matches the decrypted hash, the license is verified as authentic and unmodified.

1. Authenticity: It provides proof that the license file was generated by a legitimate source. The signature, created with a private RSA key, can be verified using the corresponding public key.

2. Integrity: The signature ensures that the license file contents have not been altered. Any modification to the file after signing will cause the signature verification to fail.

3. Non-repudiation: The use of RSA signatures prevents the signing entity from denying the issuance of the license, which is crucial for legal and administrative purposes.

The server signs the license using its private key, and the client-side validation system verifies the license, rejecting any invalid licenses. In the current implementation, RSA-512 is used with SHA-256 as the hash function.

4. Program Encryption

To protect the software from unauthorized use and reverse engineering, we employ a block-based encryption scheme for the program's executable file. This approach offers superior protection compared to simple conditional checks that can be easily bypassed.

4.1 Basic Block Encryption

Instead of encrypting the entire file as a single unit, which could make it vulnerable to complete decryption if the key is compromised, we divide the text section (containing the actual code) into multiple basic blocks [8]. A basic block is a sequence of consecutive assembly or machine instructions with the following properties:

1. Single entry point

2. Single exit point

3. No internal jumps or branches

4. Sequential execution without interruption

**Example of Dividing Code into Basic Blocks**

Consider this assembly code:

assembly

Copy code

mov eax, 1 ; Instruction 1

add eax, 2 ; Instruction 2

jmp label1 ; Jump to label1

label1:

mov ebx, 3 ; Instruction 3

cmp ebx, eax ; Instruction 4

je label2 ; Jump to label2 if ebx == eax

label2:

sub eax, 1 ; Instruction 5

**Basic Blocks:**

1. **Block 1:**

assembly

Copy code

mov eax, 1

add eax, 2

jmp label1

1. **Block 2:**

assembly

Copy code

label1:

mov ebx, 3

cmp ebx, eax

je label2

1. **Block 3:**

assembly

Copy code

label2:

sub eax, 1

Each basic block is encrypted separately, ensuring that decryption occurs only when execution enters a new block. This approach makes it significantly more challenging for unauthorized users to decrypt and analyze the entire program.

4.2 Challenges and Solutions

Dividing the program into basic blocks presents some challenges:

1. Dynamic jumping: In cases where the program jumps to an address stored in a register, it may enter the middle of an encrypted block. To address this, we preserve all locations containing dynamic jumps and handle them separately in the debugger.

2. Non-instruction bytes: The disassembler may not always correctly identify non-instruction bytes in the text section, potentially breaking the block usage. We mitigate this issue by filtering blocks that contain fewer instructions than addresses, particularly in the context of switch-case address tables.

3. Block size optimization: The number of blocks to encrypt is determined by a limit factor, which filters out blocks smaller than a specified number of bytes. This factor can be adjusted to balance security and performance.

4.3 HKDF (HMAC-based Key Derivation Function)

The HKDF process is used to derive unique encryption keys for each block of the program from the initial key generated in the license[9]. This two-step process ensures that even if the initial key is compromised, the individual block keys remain secure.

1. Extract step: This step produces a pseudorandom key (PRK) from the input key material (IKM) and a salt. In our implementation, the IKM is the initial key from the license, and the salt could be derived from block-specific information.

2. Expand step: This step expands the PRK into output key material (OKM) of the required length, incorporating additional context-specific information. In our case, this could include block addresses or other unique identifiers[10].

Our implementation of HKDF uses the following parameters:

1. Salt: We use the block address (converted to 4 bytes or 8 bytes) as the salt. This ensures that each block has a unique salt, even if the other parameters are the same.

2. Input Key Material (IKM): The initial key (key\_bytes) from the license serves as the IKM. This is the secret key material from which we derive the block-specific keys.

3. Info: The PC ID is used as the info parameter. This adds an extra layer of context-specific information to the key derivation process, binding the derived keys to the specific machine.

4. Output Length: We set this to AES\_KEY\_LENGTH to ensure the derived key is the correct size for our encryption algorithm.

The use of HKDF provides several benefits:

1. It generates unique keys for each block, enhancing security.

2. It allows for deterministic key generation, eliminating the need to store individual block keys.

3. It provides cryptographic separation between the initial key and the block-specific keys.

5. Encryption Method: AES-128 CTR Mode

We use AES-128 in Counter (CTR) mode for encrypting the blocks. CTR mode is chosen for several reasons [1] [11]:

1. It doesn't require padding, making it suitable for encrypting blocks of various sizes.

2. It allows for parallel encryption and decryption, potentially improving performance.

3. It turns the block cipher into a stream cipher, enabling the encryption of data of any length without the need for padding.

In CTR mode, a counter is encrypted and then XORed with the plaintext to produce the ciphertext. This counter is incremented for each block, ensuring that each block is encrypted with a unique keystream[12].

6. Angr Package and Its Use in the Project

Angr is a powerful binary analysis framework that provides a suite of tools for analyzing, reverse engineering, and exploiting binary programs[13]. In our project, we used Angr for several key purposes:

1. Control Flow Graph (CFG) Generation: We use Angr to create a CFG of the target binary, which helps us identify basic blocks and their relationships. The angr CGF is not devide blocks by jumping target address. it was added by our implementation.

2. Basic Block Analysis: Angr allows us to analyze individual basic blocks, including their instructions and properties, which is crucial for our block-based encryption scheme.

Problem we faced during the dividing the blocks was non instruction bytes in the text section. That was making the program entering into the middle of the block. And also, reading of this encrypted info. Angr know how to detect non instructions bytes, but its not perfect. The problem was loaded when we protected software that uses switch case. In switch case, smart compiler uses address table instead of chain of conditional jumps. To mitigate this problem, we take care to filter block that the number of address in it is bigger than the number of instructions (under the assumption it is usully will be address). but yet, handling non instructions bytes in the text sections rely hevily on the angr module to detect it.

The use of Angr significantly simplifies the process of binary analysis and manipulation, allowing us to implement our DRM system more efficiently and effectively[14].

7. Summary of the License Generation Server Side

The license generator contains (on the server side):

1. The exe file license generator (written in C) that generates the actual license.

Output: Binary file containing license fields with digital signature.

2. Encryption program - encryption of the input exe file and creation of helping file for the activation program.

Output:

a. Encrypted exe file.

b. Block ranges binary file

c. Dynamic jumps address binary file

The encrypted exe file is sent to the user as the protected system. The binary files and the license are sent with the activation exe file.

8. Activation Program

The activation program is responsible for activating the protected program. Without the activation program, the protected program is useless. To create the activation program, we developed a debugger to activate the protected exe file[15]. The program first reads the block ranges file and adds a breakpoint instruction (int3 or 0xcc) as the first instruction of each block. Due to the single entry point of the block, the breakpoint provides an opportunity to decrypt the entire block.

A few challenges we faced during this process include:

Base Relocation Table: This table contains information on all the places within the PE file where the addresses need to be adjusted.

Relocation Entries: Each entry in this table specifies:

Page RVA (Relative Virtual Address): The starting address of a 4KB page where relocations need to be applied.

Block Size: The total size of the block, including the header and all relocation entries.

Type and Offset: The type of relocation (e.g., high, low, absolute) and the offset within the 4KB page where the relocation should be applied.

Relocation Process:

Load Address Calculation: When the PE file is loaded at a different address, the difference (delta) between the preferred base address and the actual load address is calculated.

Applying Relocations: The relocation entries are processed, and the delta is added to the addresses at the specified offsets within each page. This adjusts the addresses to their correct values in memory.

Example:

Suppose a PE file has a preferred base address of 0x10000000, but it gets loaded at 0x20000000.

The delta is 0x20000000 - 0x10000000 = 0x10000000.

If there is a relocation entry for an address at 0x10001000, the loader adds the delta to this address, adjusting it to 0x20001000.

1. Reallocation addresses in the middle of the block: Since E(address + base address) ≠ E(address) + base address, the addresses in the block aren't the original ones.

Solution: At the start, we read all the reallocation addresses from the reallocation address (PE format field). After that, for each block, we check if there are addresses that are reallocated in the block. For each address, we subtract the base address from it, decrypt it, and add the base address again.

2. Dynamic jumps: As described above, dynamic jumps can jump to the middle of a basic block, which means the debugger wouldn't see the breakpoint and would think it's reading the actual instructions when in fact it's reading only the encrypted ones.

Solution: The activation program gets dynamic jump addresses in the binary file. The program will put a breakpoint for each non-encrypted call address instruction. When a breakpoint debug event occurs, there will be a check if the breakpoint address is in the dynamic jump address list, and if so, single-step mode will be set. This means a debug event will happen after the current instruction. If the instruction is encrypted (inside one of the encrypted basic blocks), after decrypting, the program will put the breakpoint on the dynamic jump instruction.

When the debugger detects single-step mode, it will check if the jump was to the middle of an encrypted basic block, and if so, it will ensure to decrypt it.

The reason that single-step mode was not set for all the instructions in the code is that it would significantly impact performance. The number of dynamic jumps into the text section is significantly smaller than the number of instructions, so using this solution for such cases is tolerable.

Footnotes:

[1] Chapter 1: Liu, Q., Safavi-Naini, R., & Sheppard, N. P. (2003). Digital rights management for content distribution. (modified)

[4] Chapter 3.2: Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of applied cryptography. CRC press. (Good)

[5] Chapter 3.2: Eastlake 3rd, D., Schiller, J., & Crocker, S. (2005). Randomness requirements for security (RFC 4086). Internet Engineering Task Force (IETF). (Good, some externals inputs)

[6] https://www.openssl.org/

[7] A DRM framework towards preventing digital piracy

Conference Paper · December 2011

DOI: 10.1109/ISIAS.2011.6122785 · Source: DBLP

CMR Technical Campus

123 PUBLICATIONS 1,959 CITATIONS

And also the paper Digital rights management: The technology behind the hype.

[8] Chapter 4.1: Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley.

[9] Chapter 4.3: Krawczyk, H., & Eronen, P. (2010). HMAC-based Extract-and-Expand Key Derivation Function (HKDF) (RFC 5869). Internet Engineering Task Force (IETF).

[10] Chapter 4.3: Krawczyk H, Eronen P (2010) HMAC-based Extract-and-Expand Key Derivation Function (HKDF). (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 5869. <https://doi.org/10.17487/RFC5869>

[11] Chapter 5: Dworkin, M. (2001). Recommendation for block cipher modes of operation: Methods and techniques (NIST Special Publication 800-38A). National Institute of Standards and Technology

[12] Chapter 5: Ferguson, N., Schneier, B., & Kohno, T. (2010). Cryptography engineering: design principles and practical applications. John Wiley & Sons.

[13] Chapter 6: Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., ... & Vigna, G. (2016, May). Sok:(state of) the art of war: Offensive techniques in binary analysis. In 2016 IEEE Symposium on Security and Privacy (SP) (pp. 138-157). IEEE. =

[14] https://angr.io/

[15] <https://www.timdbg.com/>