Kerstyne Moran

CSCI 401

Final Project

PRESENT Lightweight Cryptography Vulnerabilities

Lightweight cryptography is a subfield of cryptography that aims to provide solutions for resource-constrained devices such as sensors, health-care devices, and contactless smart cards. The way lightweight cryptography works is by maintaining a balance of security, size of device cost, and efficiency. “PRESENT is one of the first lightweight block cipher designs that was proposed for constrained hardware environments”. Its name “lightweight” originates from the creation of a low or medium-low level of security for everyday devices. Although there are many different reasons to implement lightweight, there are some debugging issues with the security of the code. In this paper, I will be discussing the vulnerabilities of PRESENT Cryptography with the help from some codes on Github. PRESENT lightweight implementation has been created through some attempts from coders but only a few have proven successful.

PRESENT Cryptography was first created by the Orange Labs (France), Ruhr University Bochum (Germany), and the Technical University of Denmark in 2007. PRESENT was designed by security experts Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. The algorithm is notable for its compact size (about 2.5 times smaller than AES). It’s based on SPN (substitution–permutation network) and is used as an ultra-lightweight algorithm for security.

The cipher works by taking a block of 64 bits and applying an 80-bit or a 128-bit key. Some lightweight block ciphers use small key sizes (less than 96 bits) for efficiency (e.g., 80-bit PRESENT). Currently, the minimum key size required by NIST is 112 bits. “Overall, it has 32 rounds, which is made up of the round key operation, an S-box layer, and a P-box layer. The key round operation takes part in the key and XORs it with the data input into the round. It then operates on 4 × 4-bit S-boxes and which considerably cuts down on processing power”. In AES, we map for 16- bit inputs to 16-bit outputs (0x00 to 0xFF).ut But for PRESENT, we have 4-bit values and which map onto 16 output values (0x0 to 0xF). The encryption is run by three different types of layers which are “Byte substitution layers: Consists of a non-linear substitution that is applied independently to each nibble of the state matrix. This substitution block is applied to 16 nibbles that complete 64 bits of information, which is the standard size of the cipher blocks.” This is the first layer to go through to change the size of the bits to make it more difficult for an “Oscar” to know how to even begin hacking an “Alice”. “ Bit Permutation (pLayer): It is a layer that mixes by means of a bitwise substitution a block of information of 64 bits, where the “i” bit of the round is moved to the position P(i).” Key expansion function (addRoundKey): PRESENT can have keys of 80 or 128 bits in length, but for this design and implementation only a key of 80 bits will be taken into account, which will be stored in a K register of that size and will be listed K_ {79} K_ {78} .... K_0. In other words, this layer is used to implement the key function to perform the loop for the key. But in each round, only the 64 most significant bits of the new calculated key will be mixed after applying the key expansion function.

The part to not forget is that the S-box substitutes a small block of bits (the input of the S-box) by another block of bits (the output of the S-box). This substitution should be one-to-one, to ensure invertibility (hence decryption) and to keep everything more secure. In particular, the length of the output should be the same as the length of the input, which differs from S-boxes in general that could also change the length.

The program I used (refer to my presentation) is credited to Petar Tonkovic on Github (3). This code is an implementation of PRESENT lightweight cryptography. Being that PRESENT is a relatively new concept, there aren't many other implementations that have been developed. It is, however, increasingly being used as the demand for the developing business/technology innovation such as medical, transportation, educational, etc.

In terms of vulnerabilities in the code, I found there were multiple times the developer used malloc in the encryption process. This could potentially be buffer-overflow because it uses malloc() function. Once buffer-overflow occurs, any attacker can launch an attack from here. The bit string is safe because it just copies one bit by one bit. The idea of a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally means that the buffer was allocated using a routine such as malloc().”Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.” In the weakness of the code, the looping of the key can cause it to overflow and crash the program.

The second vulnerability I found in the code is the use of the   get() function, which is unsafe because it assumes consistent input. They should not use get() since it has no way to stop a buffer overflow. It's seen as cautious to use, and is better to use fget() instead. When running the program, it actually gives a warning but it continues to execute. If the user inputs more data than can fit in the buffer, this will most likely result in corruption or worse. “In fact, ISO have actually taken the step of removing gets from the C standard (as of C11, though it was deprecated in C99) which, given how highly they rate backward compatibility, should be an indication of how bad that function was." The ideal way to handle things is to use the fget() function with the stdin file handle since you can limit the characters read from the user.

After performing research, the main and final vulnerability I found with this code is a *Differential Attack*. In one of my discoveries, there was a paper on differential attacks specifically on PRESENT, where rounds are implemented in order to find the key. " Firstly, we give the XORs differential distribution of S-box in[seen in presentation slides]. From the XOR's distribution table for S-box, one bit input difference will cause at least two bits output difference, which will cause two active S-boxes in the next round. Then each of the two active S-boxes will have at least two bits output difference, which will cause at least four active S-boxes in the next round." Which is, in simpler terms, a way to keep distributing to try to find the key in rounds. The differential attack will depend on the type of PRESENT, such as 16 bit or 64 bits. "As the 24 differential characteristics we found have 2 active S-boxes in the first round which are located in S-box 0, 1, 2, 12, 13 and 14, which must be non-active S-box from 3 to 11 and 15 in the first round. In each structure, the inputs in 10 non-active S-boxes can take 240 possible values, and the inputs to any two active S-boxes in each characteristic among the six active S-boxes have 224 possible values. There are 240 ∗ 2 16 ∗ 2 7 = 263 pairs for each possible characteristics, 263 ∗ 20 = 267.32 pairs satisfy 24 characteristics. Each characteristic has the possibility 2−62, so the number of right pairs is 263 ∗ 2 −62 ∗ 24 = 48 satisfying any one characteristic. For each structure, there are about 247 pairs of plaintext to be considered in total."

The output difference of 14-round differential characteristics is that there are two active S-boxes in round-15, which are x0 and x8, whose input difference is 9 and output difference will be 2, 4, 6, 8, 12 or 14. "The least significant bit of their output difference must be zero, so at most

6 bits are non-zero for the output difference of S-boxes in round 15. After the pLayer of round 15, the maximum number of active S-boxes for round 16 is 6 and the active S-boxes will be x4, x6, x8, x10, x12 and x14, and the minimum number of active S-boxes for round 16 is 2."

In a world filled with technology and new ideas every day, why do we need lightweight cryptography? I'll tell you why: efficiency of end-to-end communication.In order to achieve end-to-end security, end nodes implement the asymmetric key algorithm. For low resource-devices, e.g. battery-powered devices, cryptographic operation with a limited amount of energy consumption is imperative. Applicability to lower resource devices: the footprint of the lightweight cryptographic primitives is smaller than the conventional cryptographic ones. The lightweight cryptographic primitives would open up possibilities of more network connections with lower resource devices. Everything in the world is costly, so because the design is to be "light" we try to make less of an impact on nature while still having the power to protect everyone's personal information.

Throughout this paper, we have looked over the algorithms and basic parts of the new idea of lightweight cryptography. The idea is so new that the NIST (National Institute of Standards and Technology) has a project taking place to compete to create new light cryptography standards, which will take several years before picking the final group. They are currently in the second round with 32 additional candidates. There was a recent workshop I was able to attend on the first day. It was amazing to see something I'm currently doing research on and be able to watch it develop. But as we progress, we need to make sure low-cost lightweight solutions are secure for our small IoT (Internet of Things) devices.For these reasons, lightweight cryptography would function better to secure the sensitive data transmissions occurring every second on the small IoT.

References

(1)(2020). Retrieved 19 December 2020, from https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf

(2)(2020). Retrieved 19 December 2020, from http://www.lightweightcrypto.org/present/present_ches2007.pdf

(3)Pepton21/present-cipher. (2020). Retrieved 19 December 2020, from https://github.com/Pepton21/present-cipher

(4)(2020). Retrieved 19 December 2020, from https://eprint.iacr.org/2007/408.pdf

(5)Lightweight Cryptography | CSRC. (2020). Retrieved 19 December 2020, from https://csrc.nist.gov/projects/lightweight-cryptography

(6) CWE - CWE-122: Heap-based Buffer Overflow (4.3) . (2020). Retrieved 19 December 2020, from https://cwe.mitre.org/data/definitions/122.html