Hello there!

Thank you for your interest in Ruth’s musings in security. I aim to read one academic paper in cryptography (theory) and computer security (application) each week. Here are the musings from the week 7/1/2017 to 13/1/2017. I am very far from being an expert on these topics, therefore, if you need to contact me, to report errors in my stuff, my email address is thisemailisnotruthless@gmail.com. Also, for clarification, this is not research that was done this week. This is research that was read by Ruth this week =P

This Week in Crypto: “On the properties of the CTR encryption mode of the Magma and Kuznyechik block ciphers with re-keying method based on Crypto-Pro Key Meshing”

by Liliya R. Ahmetzyanova, Evgeny K. Alekseev, Igor B. Oshkin, Stanislav V. Smyshlyaev, Lolita A. Sonina

Premise: This article is about how to rekey a system that, for some reason or another, cannot randomly generate a new symmetric key. The paper concerns itself with CTR mode (albeit a rather specific one), and having a key that hasn’t been compromised, but has a fixed “lifetime”, and therefore needs to be changed. The authors show that one can use the decryption function of the block cipher (i.e. the cipher being used in CTR mode) and the original key to rekey the function. They then provide proofs on why this is a satisfactory solution for particular ciphers.

CTR: CTR mode is defined in a predictable way. The important thing to note here is that the authors assume that $|IV| = \frac{n}{2}$, where $n$ is the block size of the cipher. Also, the authors assume that the key length, $k$ is such that $k = 4n$. This applies to the Magma and Kuznyechik ciphers that they are developing this standard for.

The authors then define CTR as a symmetric encryption scheme, assumed to be used with a block cipher we will denote as $E$. Therefore, they take $CTR = (K, E, D)$. Here, $K = \{0, 1\}^k$.

**Algorithm $E_K(M)$**

$\begin{align*}
IV &\leftarrow \{0, 1\}^\frac{n}{2} \\
M_0 || M_1 \ldots || M_m &\leftarrow M \\
\text{For } (i = 0, 1, \ldots m)\{ \\
C_i &\leftarrow M_i \oplus E_K(IV||\langle i \rangle) \\
\} \\
\text{Return } IV || C_0 || C_1 \ldots || C_m
\end{align*}$

**Algorithm $D_K(C)$**

$\begin{align*}
IV || C_0 || C_1 \ldots || C_m &\leftarrow C \\
\text{For } (i = 0, 1, \ldots m)\{ \\
M_i &\leftarrow C_i \oplus E_K(IV||\langle i \rangle) \\
\} \\
\text{Return } M_0 || M_1 \ldots || M_m
\end{align*}$

In the authors’ description of CTR, they do not do any padding or use a compression function in the last block, and simply truncate the last $C_i$ to the length of the last (partial) block. In my definition, for simplicity, I just assumed that the message is padded to a length that is a multiple of $n$.

CTR-CPKM: The authors’ contribution is the Crypto-Pro Key Meshing (CPKM) algorithm to perform rekeying. The authors define $CTR − CPKM_l = (K, E, D)$, using the CPMK rekeying method every $l$ blocks. In these algorithms, $c_e, c_d, K_e, K_d$ are global variables, and $D_1, D_2, D_3, D_4 \in \{0, 1\}^n$ are distinct constants.

**Algorithm $E_K(M)$**

$\begin{align*}
\text{If } (K_e = \bot)\{ \\
c_e &\leftarrow 0 \\
K_e &\leftarrow K \\
IV &\leftarrow \{0, 1\}^\frac{n}{2} \\
M_0 || M_1 \ldots || M_m &\leftarrow M
\end{align*}$
For \( (i = 0, 1 \ldots m) \}{
\begin{align*}
  c_e & \leftarrow c_e + 1 \\
  \text{If} (c_e = l) \{ \\
    K_e & \leftarrow |E_{K_e}^{-1}(D_1)||E_{K_e}^{-1}(D_2)||E_{K_e}^{-1}(D_3)||E_{K_e}^{-1}(D_4) \\
    c_e & \leftarrow 0 \\
  \}
  C_i & \leftarrow M_i \oplus E_{K_e}(IV || (i)) \\
  \text{Return} \ IV || C_0 || C_1 \ldots || C_m
\end{align*}

\textbf{Algorithm} \( D_K(C) \)

\begin{align*}
  \text{If} (K_d = \bot) \{ \\
    c_d & \leftarrow 0 \\
    K_d & \leftarrow K \\
    IV || C_0 || C_1 \ldots || C_m & \leftarrow C \\
  \}
  \text{For} (i = 0, 1 \ldots m) \{ \\
    c_d & \leftarrow c_d + 1 \\
    \text{If} (c_d = l) \{ \\
      K_d & \leftarrow |E_{K_d}^{-1}(D_1)||E_{K_d}^{-1}(D_2)||E_{K_d}^{-1}(D_3)||E_{K_d}^{-1}(D_4) \\
      c_d & \leftarrow 0 \\
    \}
    M_i & \leftarrow C_i \oplus E_{K_d}(IV || (i)) \\
    \text{Return} \ M_0 || M_1 \ldots || M_m
  \}
\end{align*}

\textbf{Security Games and Proofs:}

Note: As always I do with papers with security proofs, I will not present the actual security games in their original level of detail. However, this paper is not very clear with its games, unlike previous papers that I have reviewed. I do have a writeup formalizing all the games, that you can ask me for, if you are interested.

First, define \( \text{Adv}_E^{PRF}(t, q) \) as the maximal probability that any adversary, running in time \( t \) and making \( q \) oracle queries, can differentiate between a random mapping from \( \{0, 1\}^n \) to \( \{0, 1\}^n \) and the block cipher \( E_K \) under a random key (not known to the adversary).

Second, define \( \text{Adv}_E^{PRF-CCA}(t, q_1, q_2) \) as the maximal probability that any adversary, running in time \( t \), can differentiate between a random permutation from \( \{0, 1\}^n \) to \( \{0, 1\}^n \) and the block cipher \( E_K \) under a random key (not known to the adversary). Here, the adversary is given two oracles, one that performs the function \( F \) or the encryption \( E_K \) and the second performing its respective inverse. The adversary is allowed up to \( q_1, q_2 \) queries to the two oracles, respectively.

Third, define \( \text{Adv}_E^{LOR-CPA}(t, q, m) \), where \( SE \) is a symmetric encryption scheme (e.g. CTR or CTR-CPKM) as the maximal probability that any adversary running in time \( t \), making \( q \) queries of length at most \( m \), can provide \( (M_0, M_1) \), then distinguish between \( SE(K_0) \) and \( SE(K_1) \).

Then, the authors claim that

\[ \text{Adv}_E^{LOR-CPA}(t, q, m) \leq 2 \text{Adv}_E^{PRF}(t + q + nm, qm) \]

\( \text{Adv}_E^{LOR-CPA}(t, q, m) \leq 2 \text{Adv}_E^{PRF}(t + q + \frac{n}{2} q \cdot l, \frac{k}{m}) + 2 \text{Adv}_E^{PRF}(t + mlq \cdot q \cdot l) + 2m \delta \)

where \( \delta \) is a small value, given \( q, l \ll 2^n \).

The authors proceed to apply this to the case of the Magma cipher, and approximate the PRF and PRP security using birthday attacks and brute force attacks (the only attacks that work against an ideal \( E \)). Then, they were able to show that with \( CTR-CPKM \), that a tighter bound was achieved, meaning that security has not been weakened by the addition of the rekeying, and likely was improved by a factor of \( m \).

\textbf{This Week in Security: “Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR”}

by Dmitry Evtyushkin, Dmitry Ponomarev, Nael Abu-Ghazaleh

\textbf{Premise:} In software development, it is impossible to achieve a “bug-free” environment. Therefore, hardening techniques like address space layout randomization (ASLR) were introduced as a defense against adversaries seeking to exploit these vulnerabilities. The authors provide a technique that weakens the protection that ASLR provides, which makes use of side channel information provided by the branch target buffer (BTB) to reverse engineer all or some bits of entropy used in the randomization.

\textbf{Preliminaries:} ASLR randomizes the location of key program components in virtual memory. This makes it substantially more difficult for an adversary to locate particular code segments in memory, or leverage them to launch an attack. The authors consider two kinds of ASLR. The first is Kernel ASLR (KASLR), where kernel code is segmented and their virtual memory locations are randomized. The second is user ASLR, which does the same for user-level address spaces.

\textbf{Threat Model:} The threat model for the KASLR attack is as follows:

- Assumes the attacker has control over one process running on the target system with normal user privileges (the spy process)
- Assumes that KASLR randomizes code locations at boot, as is done with the KASLR algorithm.
- The attacker can interact with the kernel through system calls
- The attacker cannot brute force the correct address (in the kernel, brute forcing will most likely crash the system)
The attacker’s goal is to recover the address of one kernel routine, such as a system call handler, in virtual memory.

The threat model for the user process ASLR is instead as follows:

- Assume the presence of two user processes, one victim process that is the target of an attack (e.g. because it has access to sensitive data) and one spy process that uses the same BTB (same virtual core co-residency).
- The attacker has complete knowledge of the algorithm behind the ASLR, but cannot retrieve the ASLR entropy bits directly, as the victim is compiled as a position-independent executable.
- The attacker’s cannot brute force the correct address (usually too many bits of entropy), and aims to reduce the entropy of the ASLR routine.
- The spy process can interact with the victim process (e.g. by initiating network connections).

**BTB Side Channel:** The BTB works in a very similar way to regular caches. It allows programs to cache the target addresses of recently executed branch instructions, and is shared between applications on the same core. Like regular caches, there is an indexing function, that determines where in the BTB a target address may be stored. Loosely speaking, when applications are writing the same locations in the BTB (this does not need the be the same location in memory, just the same location in the cache), the applications will slow down because the jump target that they need will not always be in the BTB, since it will be overwritten by the other applications. Conversely, the applications will speed up if they are able to use different locations in the BTB, since the targets they need will be in the BTB more often. Without going into excessive amounts of detail, we overview the two kinds of BTB collisions (i.e. when programs write the same location in the BTB) that the authors’ attacks leverage on.

The first is a Same Domain Collision (SDC), where two applications running on the same protection domain are writing to the same BTB. The victim will write to the BTB some jump that it is performing, and the spy will coordinate its actions and accesses to the BTB to detect collisions or lack thereof, and use this to derive information about the entropy bits used in the ASLR. In order for this to happen, SDC’s must be detectable. To demonstrate this, the authors set up a spy and victim process, and showed that, over 100,000 measurements, a SDC leads to a consistently discernable timing increase (about 9-11 cycles). In other words, when the spy and victim access the same virtual address, there is a measurable increase in time needed for the spy’s jump to occur, since the target is not available in the BTB (having been overwritten by the victim).

The second is a Cross Domain Collision (CDC), where kernel code and the spy process are writing to the same BTB, resulting in collisions. The reason this works is because the BTB address indexing function does not make use of the full virtual memory address, because, if it did, there would be no collisions (the kernel and user code are not stored in the same virtual spaces). The authors reverse engineered a Haswell processor to show that some bits were not being used for BTB addressing, showing how CDC’s can occur: only bits 0 to 30 of the virtual addresses were being used in the function, making collisions frequent (the addresses are 48 bits long).

**Attack on KASLR:** When KASLR is enabled, a sequence of random bits are generated during the boot process that will provide a random offset. All physical and virtual addresses are offset by that amount. Therefore, if the adversary can calculate this offset, the effects of KASLR are completely nullified. To carry out the attack, the authors located a branch instruction in the kernel that the spy process can easier trigger. For this, the authors chose to use the command to open a file, with an erroneous file name, since this is fast, and located at a predictable location. The Linux kernel studied only randomizes 9 bits of the virtual addresses, therefore, the authors just needed to try 512 different addresses in order to generate a collision. The spy process initiates a jump that would cause a collision with each of the possible addresses of the jump instruction chosen above. Based on the time taken to return, the spy process was then able to retrieve all 9 bits of entropy used in the KASLR. This attack ran in 60 milliseconds, and the correct solution was very obvious from the timing data.

**Attack on ASLR:** One challenge of the attack on User ASLR is trying to ensure that the spy and victim process were accessing the same BTB, which means that they needed to run on the same core. In the case of a hyperthreaded core, the applications need to run on the exact same thread. The authors therefore did some research into way to ensure this would happen. The authors tried one method where the OS scheduler was manipulated using commands, as well as through the use of dummy programs. The other method involved running multiple copies of the spy process, then using information that the system logs to determine which spy process was running in the desired environment. Both of these methods were successful.

When the authors wrote a proof-of-concept of this, they realized that because the Haswell processor does not use the 30th and higher bits for BTB addressing, and the ASLR in Linux randomizes bits 12 to 40, only 18 bits can be retrieved. However, this greatly reduces the entropy. This was conducted in almost the same way as the KASLR attack, so I will leave out the details of it. The authors’ implementation retrieved 8 bits, testing 100 addresses per second.

**Mitigation of Attacks:** The authors first consider possible software mitigations. Software countermeasures are
limited because they are not able to control how branches are mapped to the BTB entries, thus they do not address the root cause of the side channel. The authors proposals are either to have more fine-grained ASLR schemes (beyond just a random offset), or making accurate time readings difficult. In KASLR, very limited entropy is used, with the assumption that an adversary trying to brute force the solution will crash the system, and therefore will not be able to try have many guesses before a reset. This paper shows that this is erroneous. The authors show that the number of bits could be increased to 17 (from 9) without significant overhead, which would strengthen the system quite a bit.

Then, the authors consider possible hardware mitigations. A hardware solution that would fundamentally mitigate BTB-based attacks is to change the BTB addressing mechanism in a way that prevents exploitable collisions in the BTB. The attack against KASLR can be mitigated by using the full virtual address for accessing the BTB, eliminating collisions between the user and kernel code. Alternatively, different indexing functions can be used in user and kernel level code (e.g. adding a secret constant, generated at boot, to the existing BTB hash in kernel mode). Similarly, in protecting user ASLR, different processes can use a different secret constant.