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 25/6/2016 to 1/7/2016. 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 thisemailisnottruthless@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:
“Low Probability Differentials and the Cryptanalysis of Full-Round CLEFIA”

By SAREH EMAMI, SAN LING, IVICA NIKOLIC, JOSEF PIEPRZYK AND HUAXIONG WANG

Premise: CLEFIA is a block cipher designed by Sony. It has a Feistel structure and a 128-bit key and it can be proven that there is a upper bound of $2^{-128}$ chance that an exploitable related-key differential can be found.

Key Results: The authors do not disprove that there is a low chance of an exploitable key relation, however, they do present a class of weak keys. This is significant because it shows that weak keys can be found, even though the probability of chancing upon them is extremely low. Specifically, they found $2^{14}$ triples of $(K, K', D)$, a key pair and a constant, which satisfy the following conditions for any plaintext $P$:

$$K = K' \oplus L_1(D)$$

$$E_K(P) \oplus E_{K'}(P \oplus L_2(D)) = L_3$$

Where $L_i$ are fixed linear functions, which can be found in the paper. The authors found this by noting that between rounds, the key schedule generates keys that are a linear transformation of the key from the last round. They also note that the Feistel structure ensures that the state update function was iterative. Therefore, when the round key change exactly cancels out the state update function, the property will propagate across all the rounds of the cipher, meaning the attack has the same complexity for any number of rounds in the cipher. In addition, the authors note that this attack easily extends to any cipher with a state update function is maintained between rounds (but can be an arbitrary function) and linear derivation of round keys from each other (but there can be non-linear permutations applied to the master key before this key derivation).

Implications: The authors show that given $(K, K')$, that there is a membership test for the class of keys that has complexity $2^8$. This simply makes use of the above property, while avoiding exhaustively searching for $D$.

The authors also show that finding a weak key pair can be done with complexity $2^{122}$, which is faster than the generic $2^{128}$.

They also discuss the implications of a weak key pair on hash collisions and CLEFIA in hashing mode. They show that with control over the key (to change it to a weak one), an adversary will ensure that hash collisions exist.

Overall, the authors do not reveal a weakness in CLEFIA on a practical level, and don’t actually even give an example of a weak key, since the complexity of finding them is high. However, on the theoretical level this affects how we should think about related key attacks in cipher design, since a low probability of finding a random weak key does not mean that one cannot be found through inspection of the cipher. This is also a good argument against “simplistic” round functions and key schedules that are identical round to round.

This Week in Security:
“Too LeJIT to Quit: Extending JIT spraying to...
ARM" by WILSON LIAN, HOVAV SHACHAM AND STEFAN SAVAGE

Premise: The authors present an attack strategy against RISC architechtured systems and demonstrate this on WebKit’s JavaScript-Core JS engine on ARM. This attack extends JIT attacks from x86 systems (CISC) to RISC systems. Systems typically make use of DEP (Data Execution Prevention) and ASLR (Address Space Layout Randomization) to prevent code injection and code reuse attacks. That’s a lot of big words so here are are bunch of definitions:

1. Code Injection Attacks: This is when an attacker corrupts data in memory to execute his own code on the victim’s system. An example of this would be corrupting the instruction pointer to point to an area containing data instead of instructions. Then the system will interpret the data as instructions and execute malicious code.

2. DEP: This is a protection against code injection attacks, marking pages as memory so it cannot be executed as instructions.

3. Code Reuse Attacks: This makes use of code that is already in the victim’s system to execute malicious code.

4. ASLR: This is a protection against code reuse, by randomizing the location of instructions in memory so that the attacker will struggle to redirect control to his desired functions (or “gadgets”).

5. Gadget Chaining: This is when individual portions of existing instructions are used together by the attacker maliciously.

6. JIT: A Just-In-Time compiler, which ensures better performance by allowing for more dynamic recompilation and flexible interpretation. However, because this is compiled and run at run-time, the instructions and data used by it does not have the protection of DEP and ASLR.

7. Spraying Attacks/ NOP sleds: A NOP is an operation that does nothing. Therefore, if an attacker wants to redirect the control of code to his malicious code, he will “spray” the instruction stack with many NOPs (a NOP sled) followed by malicious code. Therefore, if control ever goes to any of the NOPs, the control will eventually get to the malicious code and run it in entirety.

With all that out of the way, the attack here makes use of JITs being excused from the protection of DEP and ASLR. Therefore, through JIT spraying, code injection and code reuse attacks can still be feasible. In fact, JIT spraying can control up to 4 out of 5 bytes in executable memory of the JIT, thereby defeating DEP.

Prior attacks were done on CISC systems, which made use of an instruction set architecture that was more complex, and therefore more malleable. This allows valid data to be read as valid executable instructions. In RISC, this is much harder because of the simpler instruction set architechture. The authors get over this hurdle by, among other things, making use of RISC systems that allow both “Thumb” and “ARM” instruction sets. This allows some malleability in the instructions which the authors were able to exploit. The authors also made use of code reuse attacks, instead of code injection attacks, since code injection is challenging due to resynchronization. These code reuse attacks then make use of gadget chaining to attack the system. Finally, they make use of predictability in memory allocation of executable memory of the JIT to target the JIT spraying.

In the proof of concept attack by the authors, the authors were successful about 35% of the time in reading memory of the victim. Assuming a corrupted function pointer, they were able to pinpoint gadgets, spray instructions so that there is a high probability it will be accessed by the program and made use of existing of a vulnerability in the ReadByte() function in the WebKit to point to arbitrary attacking code.

They provide mitigations to attacks like the one they conducted. They present existing mitigations: random NOP insertions which will remove the predictability of the location of code to reuse and blinding of fields so that attackers cannot predict register values. They propose additional measures: Register randomization and JIT allocation randomization (position of functions on a page).