This Week in Security

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 2/7/2016 to 8/7/2016. 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:

"How to Construct Random Functions"

by Oded Goldreich, Shafi Goldwasser and Silvio Micali

Key Definitions: In this paper, the authors discuss a theory of randomness for functions. This is a theoretical evaluation when a single, or a family of pseudorandom functions are presented (e.g. Keyed hash functions). First, they recap various definitions for the randomness of a string, selected from a set S of strings. The definition they choose to go off of is this following:

Definition A set S is polynomial random if programs that run in polynomial time lead to identical results when fed either with elements randomly selected in S or with elements randomly selected in the set of all strings.

So what does this mean for functions?

Definition If no polynomial time algorithm, asking for the values of the functions at arguments of its choice can distinguish a computation during which it receives true values of the function, from a computation during which it receives the outcome of independent coin flips, the function is polynomial random.

Primitives: The authors’ work is based on accepted notions of one-way functions and cryptographically-strong pseudorandom bit generators (CSB generators).

While a one-way function is, by definition, computationally infeasible to invert, this does not mean that it is random. For example, one-way functions based on RSA or DLP have algebraic properties that allow randomness to be compromised, with regards to the above definition. More specifically, in the case of RSA, note that \((x^{-1})(y^{-1}) = (xy)^{-1}\), meaning that if RSA itself is used as a random function, its behavior on certain inputs can be predicted from other inputs, making this function distinguishable from a truly random function. Therefore, the important thing to note is that if \(f\) is selected in a poly-random collection, not only the value of \(f\) at argument \(x\) cannot be computed from the values of \(f\) at other arguments, but it cannot even be recognized when given.

A CSB generator is an efficient (polynomial time) deterministic program to stretch a \(k\) bit long input seed to a \(k^t\) bit long output sequence. The definition adopted by the authors (from work by Blum and Micali) of a cryptographically strong CSB is one that passes the next bit test. Loosely speaking, this means that the probability that given a string of bits, one can statistically determine the next bit (the probability of it being 1 or 0 is sufficiently far from \(\frac{1}{2}\)), is low. More specifically, this is the definition proposed:

Definition For a polynomial \(P\), let \(S\) be the set of all \(P(k)\) bit long sequences, for all \(k\). Let next-bit test \(T\) for \(S\) be a polynomial time algorithm that given \(k\) and the first \(i\) bits of output for \(k\) (it does not get the value of the seed \(x\)), outputs a bit \(b\). The probability that that bit is equal to the next bit of the sequence is given by \(p_i^k\). \(S\) passes the next-bit test if for all polynomials \(Q\), for all sufficiently large \(k\) and all integers \(i < P(k)\), we have \(|p_i^k - \frac{1}{2}| < \frac{1}{Q(P)}\).

It has been shown by Levin that:
Theorem 1 There exists a one-way function if and only if there exists a CSB generator:

Contributions: The authors adapt the definition of statistical tests for strings to one for functions. Instead of being given a string, \( T \) is now given oracle access to a function. The oracle must then output a bit 0 or 1, and the probability that this bit is correlated to whether the function belongs to a particular family of functions or not must be bounded by \( \frac{1}{\sqrt{k}} \). More specifically:

Definition A polynomial time statistical test for functions is a probabilistic polynomial time algorithm \( T \) that, given \( k \) as input and access to an oracle \( O_f \) for a function \( f : \{0, 1\}^k \rightarrow \{0, 1\}^k \), outputs either 0 or 1. Algorithm \( T \) can query oracle \( O_f \) only by writing on a special query tape some \( y \in \{0, 1\}^k \) and will read oracle answer \( f(y) \) on a separate answer tape. \( O_f \) prints its answer in one step. Let \( F = \{F_k\}_k \) be a collection of functions. We say \( F \) passes the test \( T \) if for any polynomial \( Q \), for all sufficiently large \( k \), \( |p_k^f - p_k^H| < \frac{1}{\sqrt{k}} \) where \( p_k^f \) denotes that probability that \( T \) outputs 1 on input \( k \) and access to function oracle for \( f \in F_k \) and \( p_k^H \) is defined similarly, but for \( f \in H_k \), the set of all random functions.

The authors give a construction of a function \( F \) which uses a CSB generator to construct a poly-random collection of functions. Using a CSB generator that stretches a seed of length \( k \) to a bit sequence of length \( 2k \), \( G \), let \( G(x) = b_1^x \ldots b_{2k}^x \), for seed \( x \in \{0, 1\}^k \). Recall that when \( S_k \) is the multiset of \( 2k \) bit sequences output by \( G \) on seeds of length \( k \), we have that \( S = \bigcup_k S_k \) passes all polynomial time statistical tests.

Then, take \( G_0(x) = b_1^x \ldots b_k^x \) and \( G_1(x) = b_{k+1}^x \ldots b_{2k}^x \). Now, given a binary string \( a_1 \ldots a_t \), define

\[
G_a(x) = G_{a_1}(\ldots (G_{a_t}(G_{a_1}(x)))\ldots )
\]

Then, define family of functions \( F_k \) such that \( f_k(y) = G_a(x) \) for all \( k \in \{0, 1\}^k \). Thus, \( F = \{F_k\}_k \) is the collection functions over all possible \( k \).

The authors provide a proof that if \( F \) is a collection as defined above, and \( G \) is a CSB generator passing all polynomial-time statistical tests for functions, then \( F \) passes all polynomial-time statistical tests for functions. Recall that there exists one way functions if and only if there exists a CSB generator of the form \( G \). Therefore, we have that:

Theorem 2 If there exists a one-way function, then there \( F \) collection of functions that passes all polynomial time statistical tests.

Finally, the authors introduce the concept of “inference” in polynomial time. This is to aid more practical applications, such as measuring the randomness of various functions with regards to how hard they are to distinguish from random. In particular,

Definition Let \( A \) be a probabilistic polynomial time algorithm. \( A \) is given access to oracle \( O_f \), from before, to query. \( A \) then chooses \( x \), an input value to \( O_f \) that he has not queried the oracle with. \( A \) is cut off from \( O_f \) and presented with \( f(x) \) and a random string \( y \). We say that \( A \) passes the exam if it correctly guesses which is \( f(x) \). We say that \( A \) -infers the \( F \) collection if input \( k \), for infinitely many \( k \), it passes the exam with at least probability \( \frac{1}{2} + \frac{1}{\sqrt{k}} \). This is taken over all \( f \in F_k \) and all random strings \( y \).

Definition A collection of functions \( F \) can be polynomially inferred if there exists a probabilistic polynomial-time algorithm \( A \) that \( Q \) infers \( F \).

This Week in Security: “ROP is Still Dangerous: Breaking Modern Defenses” by Nicholas Carlini and David Wagner

Premise: Return Oriented Programming (ROP) is a memory-safety attack. This is basically a form of “gadget chaining”, where existing code in the system are executed by the attacker to some malicious end. The authors show that it is possible to conduct ROP attacks that defeat many existing ROP defenses.

Attack Primitives: The authors present three attack primitives:

1. ROP attacks defer from normal instructions by deviating from the pattern of having ret instructions return back to an instruction that immediately follows a corresponding call. The authors create ROP attacks that evade defenses that check whether ret instructions target an instruction that immediately follows some call.

2. The authors create attacks that specifically evades the defenses used, at runtime to classify “gadget” execution from “normal” execution of instructions. This allows “gadgets” to run undetected.

3. The authors exploit the fact that some defenses maintain only a limited amount of history and then inspect this. The history can then be cleansed by the malicious program prior to when a check is slated to happen.

Here are their respective solutions:

1. Use only call-preceeded gadgets. By allowing gadgets to be more complex than they are traditionally, the authors found sufficient usable binary code to mount entirely call-preceded ROP attacks, by accepting gadgets that are longer, or contains jumps.

2. Most runtime monitoring algorithms classify gadgets and non-gadgets by length, and will flag a ROP attack when there are beyond a threshold of short
segments. The authors make use of benign looking longer gadgets, which do not affect the running of the malicious instructions. This fools more runtime defenses.

3. Since history inspection defenses cannot continuously monitor execution without large overhead. Therefore, existing defenses only inspect the history at predictable times (e.g. when a syscall is made). Therefore, malicious programs can insert no-ops when the inspector is running, or overwrite records of recent calls and jumps.

**Application:** The authors defeat two commonly used defenses against ROP. The first is kBouncer:

- **A history hiding attack.** kBouncer maintains a record of the 16 most recent indirect jumps in a Last Branch Record (LBR) and inspects this before a syscall. Since an ROP attack has the objective of issuing a syscall (to allow the attacker to remove the memory protection), the environment needs to be prepared to hide the ROP attack prior to the malicious syscall. First, to flush the LBR, a call-preceeded gadget that will perform a ret (which will be recorded in the LBR) needs to be performed 16 times, so that all the jumps captured in the LBR are innocuous (i.e. call-preceded). Then, a long termination gadget which needs to be at least 20 instructions long, and minimally affects register/stack values. This is so that kBouncer sees one “non-gadget” in the last 20 instructions. Then gadgets need to restore the stack and register values for the syscall to be successful.

- **An evasion attack.** The evasion attack is similar to the history hiding attack, except that even if kBouncer maintained a record of all the calls ever made, the evasion attack would still work. This is because all gadgets in the evasion attack are call preceeded. In doing this, the authors show the power of existing gadgets, even when we just exploit the call-preceded ones.

The second system attacked is ROPecker. ROPecker was built on ideas from kBouncer by introducing more sophisticated heuristics to detect ROP attacks. However, these can still be attacked as in kBouncer:

- **Repeated history hiding attack.** ROPecker only marks a few pages at a time as executable, and will run the ROPecker inspection whenever a page fault occurs (need to access a new page), or when a syscall is made. However, this is entirely predictable. The attack therefore needs to alternate between phases of “loading” useful pages into the executable set, “attacking” to invoke these gadgets and “flushing” to hide history when the attacker knows an inspection is coming up. This is a more intricate version of the attack against kBouncer, where the page loads must have termination gadgets (i.e. long gadgets) called prior to them, so ROPecker will not look back in the history past this termination gadget. After the page load gadget, we use another termination gadget so ROPecker only looks back to the preceeding termination gadget, and forward to the upcoming termination gadget. By only loading a small number of pages at a go, the attacker can completely evade ROPecker. Prior to the syscall, a similar flushing of the LBR, termination gadget and restoration of registers as in the attack against kBouncer must be used. Finally, between inspections by the ROPecker, the attacker must make small sets of computations at a go, and save its own state before the flushing and termination gadgets potentially clobber register and stack values.

- **Evasion attack.** The evasion attack, similar to the case of kBouncer, is resistant to arbitrary checks from ROPecker, by being more selective with gadgets used and the order they are used in. Even if ROPecker only allows one page in its executable set, or makes checks at unpredictable times. This simply involves using call-preceded gadgets, as before. In addition, every 10 gadgets, the termination gadget must be called. This is because ROPecker also checks for a long sequence of short segments (between 11 and 16 in existing implementations).