CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Pseudo-Random Numbers Prediction
Student:	Richard Molnár
Supervisor:	Ing. Filip Štěpánek
Study Programme:	Informatics
Study Branch:	Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2018/19

Instructions

Random numbers are generated by using the so-called pseudo-random number generators (PRNG). These deterministic algorithms use a seed to generate a sequence of numbers that satisfy the parameters on randomness. The sequence can be predicted in case the seed value is known. The goal of the thesis is to design and implement an universal tool for cracking common PRNGs used by web applications.
Tasks:
- Analyze the PRNG algorithms used by PHP and web applications. Focus on algorithms that are not considered secure by current standards.
- Analyze existing solutions for cracking of these PRNG algorithms.
- Design and implement a tool for discovering the seed of given PRNG sequence. The tool will be used by security specialists or penetration testers.
- Verify the functionality of the tool by using it to exploit PRNG weakness of the applications running in a laboratory environment (open software instances).
- Discuss the results.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 2, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

Pseudo-Random Numbers Prediction

Richard Molnár

Supervisor: Ing. Filip Štěpánek

16th May 2017

Declaration

In Prague on 16th May 2017 ....

Citation of this thesis

Molnár, Richard. *Pseudo-Random Numbers Prediction.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Táto práca sa všeobecne zameriava na tému generátorov pseudonáhodných čísel so zameraním na PHP. V úvode práce je rozobraná analýza existujúcich generátorov pseudonáhodných čísel a ich využitie a implementácia v praxi. Ďalej sú hodnotené rôzne techniky na prelamovanie týchto generátorov. Úlohou implementačnej časti práce je implementovat algoritmy na prelamovanie generátorov číselných rád a súčasne vyvinút sadu nástrojov na podporu tejto činnosti. Tieto nástroje budú využívat novú techniku na prelamovanie za pomoci dúhovych tabuliek. Následne sú porovnané existujúce programy s nástrojmi vyvinutými v tejto práci, aby boli preukázné ich schopnost.

Klíčová slova pseudonáhodost, generátory náhodnych cisel, PHP, bezpecnost, Mersenne Twister, .NET

Abstract

This thesis focuses on the topic of pseudo-random number generators (PRNG) in general with focus on PHP. This work starts by analyzing existing random number generators and their implementation in real world. It continues by assessment of several techniques to crack PRNGs. The implementation goal of this thesis is to implement current techniques for PRNG cracking and to develop a set of tools to make this task easier by implementing supporting tools. The toolkit also makes uses of new PRNG cracking technique with a use of rainbow tables. Existing software for PRNG cracking is later explorer and performance benchmarks made to showcase abilities of newly developed toolkit.

Keywords pseudo-random number generators, PHP, security, Mersenne Twister, .NET

Contents

List of Figures

List of Tables

Introduction

Random numbers are number sequence that are part of almost every computer program. They are used to randomly initialize variables, to randomly perform some actions or to simulate AI behavior. But random numbers are often used where the next random number should be absolutely unpredictable – in lotery numbers, cryptographic tokens or password generation. Generators that satisfy this requirement are called cryptographically secure.
Plenty of pseudo-random number generators (PRNGs) satisfy this condition. However, in reality, difference between secure and insecure generators is often not stressed enough. Almost every software platform offers insecure PRNG by default and the programmer needs to know that he needs the secure one. But programmers often don't know this difference or don't pay too much attention to it, which can introduce new vulnerabilities to the system.
The job of security penetetration testers (or pentesters) is to asses the security of the given software. They usually check for variety of vulnerrabilities such as SQL injection or XSS. Vulnerabilities of these types can be easily exploited by powerful toolkits available for pentesters.
PRNG vulnerabilities don't have this luxury. Some tools exists, but they are far from widespread, hard to use and very slow. This has a consequences that pentesters rarely test these vulnerabilities a they are left in the system. A highly motivated hacker can then exploit these even though the system passed all the ussual penetetration tests.
This thesis starts first by exploring the topic of pseudo-random number generators. Many of their properties are explained, which is going to be critical for understanding the exploits.
Next, several popular software platforms are listed with their respective PRNGs. The analysis chapter ends by showing several existing software solutions for PRNG cracking.
In the Design chapter, I focus on designing PRNG cracking toolkit based on the knowledge from the Analysis chapter. Main focus is on simplicity of usage and flexible software architecture. The final toolkit consists of several tools,

all of which can be used without exiting the program.
The toolkit uses existing techniques for PRNG cracking, but during the implementation process a new technique was discovered, which makes the whole toolkit much faster.
In the end, the whole result of implementation is tested and compared against other software solutions.

Chapter 1

Analysis

In this chapter, I will first start with definition of random numbers, how are they used in computers, the way of generating them and some of their properties. Then I will move to the way they are actually implemented in different software environments. I will finish this chapter with common ways of cracking random number generators and existing algorithms of doing so.

1.1 What are random numbers

Although the word random is often used in everyday language, it it pretty hard to define, what randomness actually is. Oxford dictionary defines it with the following definition :

“randomness[noun]:The quality or state of lacking a pattern or principle of organization; unpredictability.”[2]

“A random sequence is a vague notion... in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians”[3]

In computers and computer programs, there are two important concepts for this thesis : Random Number Generator and Random Number Sequence. Random Number Generator (or RNG for short) is a device that produces a sequence of numbers that meets certain statistical requirements for randomness.[4] Random numbers are essential part of almost every computer program – they are used for generating passwords, encryption tokens, in games or randomly generating a new profile picture for a forum user.

1.2 Problem with random numbers in computers

Processors have generally big problems with generating random numbers. Every CPU in every computer should be and usually is deterministic, which means that every execution of algorithm with equal inputs will have equal outputs – which is the opposite of Random. That means there is no magical algorithm that can generate random numbers on its own. This problem has two solutions: introduce a real source of randomness – entropy, or try to trick the user and produce number sequence, which only looks random, but actually isn't.

1.2.1 Problematic random numbers

The obvious solution for random numbers is to get some form of "entropy" source. In this context (as the word "entropy" usually has many meanings), entropy is some form of noise which can be used as an input for the RNG. For computer applications, the usual sources are atmospheric noise, noise from the microphone, ping time to a web server, spin time of the hard-drives, temperature of the CPU and so on[5].
Although these are easily accessible and generally good quality sources, they are rarely used – every one of them requires specific hardware or software, which doesn't have to be present on other computers. This makes the computer program less portable and less appealing to programmers, especially when they don't actually need real random numbers.

1.2.2 Pseudo-random number sequences

Another solution for the problem of generating random numbers is not to create random numbers at all. Instead, random-looking numbers can be produced – these are called pseudo-random numbers. Let's consider this number sequence :

$$1, 6, 3, 7, 9, 10, 5, 8, 4, 2$$

If a random observer would be asked to tell us if there is any order in this sequence, the obvious answer would be that this sequence has no order and thus is random. The truth is actually the opposite of this, since the previous sequence has this exact formula:

$$a_n = a_{n-1} * 6 \mod 11$$

Is this sequence unpredictable ? Far from it, but the knowledge of underlying generator is required, which can be hard to guess. This was pretty simple example, but in real-world cases, the algorithms used can be much more advanced.

1.3 Qualities of random numbers

Pseudo-random numbers can be judged from few different viewpoints. First category of tests are statistical tests, which judge random numbers in their mathematical and statistical similarity to real random numbers. Second viewpoint is security - how well the number sequence can be predicted, if previous outputs are known to an attacker.

1.3.1 Die hard tests

To test the quality of the RNG, the so called "Die hard tests" can be used. They are statistical test created by George Marsaglia in 1995 to test the statistical properties of random sequences[6]. Many pseudo-random number generators (PRNGs) can output sequences with the same statistical properties as real RNGs, making them suitable for a broad range of applications, such as Monte-Carlo simulations or randomly selecting movements of an AI enemy in a video-game. While most PRNGs easily pass these tests, they can still still be easily predictable, which the Die hard tests don't account for.

1.3.2 Security tests of random numbers

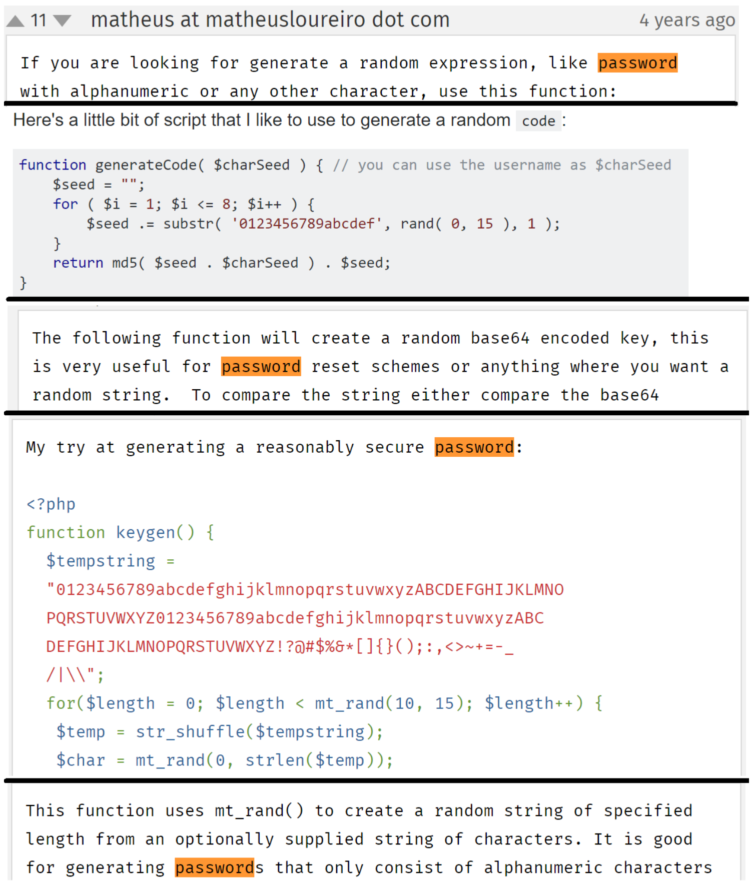
In some cases, it is required that any attacker observing the output of the PRNG will not be able to predict the next outputs based on his knowledge of the previous numbers. Almost any advanced PRNG has an internal state – a set of variables which are not visible outside of the algorithm. This internal state is used to create outputs. If the attacker can guess the algorithm that produces the random numbers and also the internal state of the generator, the attacker can predict all the future outputs, so secure algorithms cannot leak the internal variables.

1.3.3 NIST

In 2010, the National Institute of Standards and Technology has released a standardized test suit to test PRNGs from a variety of perspectives for various uses[7]. This is also the first test suite to test PRNGs for suitability in cryptographic applications. The test suit is available on-line, and generally all PRNGs that pass these tests can be considered secure by modern standards.

1.3.4 Periodicity

Every PRNG which is using only internal state to produce outputs (no more entropy is entered into it) has a "period" - a number of outputs after which the output will start to repeat. Ideal PRNGs can have periods as large as number of different internal states. Since period is bounded by the state "size", some



11 matheus at matheusloureiro dot com 4 years ago

```
If you are looking for generate a random expression, like password
with alphanumeric or any other character, use this function:
```

Here's a little bit of script that I like to use to generate a random `code`:

```
function generateCode( $charSeed ) { // you can use the username as $charSeed
    $seed = "";
    for ( $i = 1; $i <= 8; $i++ ) {
        $seed .= substr( '0123456789abcdef', rand( 0, 15 ), 1 );
    }
    return md5( $seed . $charSeed ) . $seed;
}
```

```
The following function will create a random base64 encoded key, this
is very useful for password reset schemes or anything where you want a
random string.  To compare the string either compare the base64
```

```
My try at generating a reasonably secure password:

<?php
function keygen() {
  $tempstring =
  "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNO
  PQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyzABC
  DEFGHIJKLMNOPQRSTUVWXYZ!?@#$%&*[]{}();:,<>~+=-_
  /|\\";
  for($length = 0; $length < mt_rand(10, 15); $length++) {
   $temp = str_shuffle($tempstring);
   $char = mt_rand(0, strlen($temp));
```

```
This function uses mt_rand() to create a random string of specified
length from an optionally supplied string of characters. It is good
for generating passwords that only consist of alphanumeric characters
```

Figure 1.1: Bad uses of insecure PRGN

PRNGs will start to repeat after 32768 values (15-bit), while some, such as MT19937, can have extremely large periods, in this case $2^{19937} - 1$.[8]

1.4 State of PRNG/RNG usage in reality

Although many secure solutions are already available, insecure PRNGs are often used where secure ones should be implemented, as seen in figure 1.1. On almost all software platforms the default PRNG is not cryptographically secure and many programmers don't realise this and use them, creating new vulnerabilities in their applications. These vulnerabilities can often be exploited and that is going to be the main focus of this thesis.

1.5 Common PRNG algorithms

There are many ways to generate pseudo-random numbers, but for use in practice there are few criteria which are important. Since computers always had problems with enough resources, real world applications must be efficient and fast. PRNGs are also a field of computer science very few people are familiar with, so only very few generators are actually used in reality. These generators have age-proven properties and satisfy many requirements of software developers.
Bellow are listed two most used PRNGs today.

1.5.1 Linear congruential generator

Linear congruential generator is any RNG defined as:

$$X_n = aX_{n-1} + c \mod n$$

where a is multiplier, c is increment and n is modulus. X_n and X_{n-1} are current and previous outputs of the generator.
LCG is probable the simplest PRNG in use. Since it has very small state size, it doesn't take a lot of memory and puts very little stress on CPU cache. Each output also needs only few integer operations, so it can output numbers fairly quickly.
Disadvantages are extremely low period sizes, which are at best as large as state size. Most implementations use only 32-bit state size, which means that this generator inevitable repeats after 2^{32} outputs. A solution to this problem can be larger state, however a problem with performance can easily occur - most CPUs don't support integer operations larger than 32-bit, so much slower vector operations are required. Table 1.1 is a list of state sizes vs software platform. The table also shows how popular LCG is.

This generator is also really bad from security point of view as will be discussed in later chapters.
Another issue can be high periodicity when plotted in some dimensions, which can cause problems in Monte Carlo simulations[9].

1.5.2 Mersenne Twister

It is by far the most widely used general-purpose PRNG. The most commonly used version of the Mersenne Twister algorithm is based on the Mersenne prime 2^{19937}-1[10]. The standard implementation of that, MT19937, uses a state of 624 32-bit integers.
MT has the following merits:

- Far longer period and far higher order of equidistribution than any other implemented generators. (It is proved that the period is 2^{19937}-1, and 623-dimensional equidistribution property is assured.)

Environment	State size in Bits
Borland C/C++	32
glibc (used by GCC)	31
ANSI C:, Digital Mars, CodeWarrior, C99, C11	31
Borland Delphi, Virtual Pascal	32
Turbo Pascal	32
Microsoft Visual/Quick C/C++	32
Microsoft Visual Basic (6 and earlier)	24
RtlUniform from Native API	31
Apple CarbonLib, C++11's minstd_rand0	31
C++11's minstd_rand	31
MMIX by Donald Knuth	64
Newlib, Musl	64
VMS's MTH$RANDOM	32
Java's java.util.Random	48
POSIX rand48, glibc rand48	48

Table 1.1: Common Software Environments and LCG state sizes[1]

- Fast generation.
- Efficient use of the memory. (The implemented C-code mt19937.c consumes only 624 words of working area.)

For Mersenne Twister, two functions are usually implemented. Seeding – which initializes the state according to the seed value, and generating a random number. Each random number generation takes a single integer from state, does a little bit of bit-shifting and then outputs the number. After all 624 integers from state are exhausted, Mersenne Twister performs "reload" operation - takes the integers in the state a performs a "twisting" operation between them, producing a new set of state numbers[8].

1.5.3 Default PRNG algorithms by software platforms

Since PRNGs are so essential in software applications, some form of PRNG is included in every software development environment. Bellow is a list of few popular software platforms and their basic PRNGs.

1.5.3.1 Java - java.util.Random

- Linear congruential pseudorandom number generator[11], as defined by D. H. Lehmer and described by Donald E. Knuth[12]
- The class uses a 48-bit seed and state, which is modified using a linear congruential formula.

- It is not producing secure random numbers, and is very easy to crack as will be shown later, but Java does provide secure PRNG - `java.security.SecureRandom`

1.5.3.2 .NET - System.Random

- Subtractive random number generator algorithm[13], this algorithm comes from Numerical Recipes in C (2nd Ed.). The current implementation of the Random class is based on a modified version of Donald E. Knuth's subtractive random number generator algorithm.
- State : 56 * 32-bit int
- Does not provide secure random numbers, but .NET also offers secure solution by default - `RNGCryptoServiceProvider`

1.6 PRNGs in PHP

PHP has more than one commonly used PRNG, this is the list of all generators that are easily accessible in PHP. There are many others available for PHP, but in the form of plug-ins, so their market share is orders of magnitude lower and they won't be discussed in this thesis.

1.6.1 lcg_value()

Rarely used PRNG based on combined linear congruential generator. From docs: "lcg_value() returns a pseudo random number in the range of (0, 1). The function combines two CGs with periods of 2^{31} - 85 and 2^{31} - 249. The period of this function is equal to the product of both primes."[14]
Using the search function on GitHub we can see that it returns only 14,000 results compared to 336,000 for *openssl_random_pseudo_bytes* and 2,700,000 for *mt_rand* and so this function will not be discussed further.

1.6.2 random_int() / random_bytes()

New functions introduced in PHP7[15]. Since they are available only from PHP7 and later which was released recently in December 2015, almost no applications use them yet. They are also declared and considered cryptographically secure with no known vulnerabilities, so they will also not be discussed further.

1.6.3 openssl_random_pseudo_bytes()

This function is available since PHP 5.3 and in most cases generates cryptographically secure sequences (this is not guaranteed) . This function is also not frequently used – many developers don't know about it and often it's not included in the PHP environment since it's an OpenSSL extension.

1.6.4 rand()

Most used PRNG function alongside with mt_rand(). This function has many implementations – rand() has no guarantee what implementation will be used during runtime, since this depends on PHP version, host operating system and environment variables.[16]

- If the PHP 7.1 and higher is used, rand() is alias for mt_rand(). However, PHP 7.1 is rarely used since it was release only a few months ago(December 2016) and majority of web servers still run much older PHP 5 or more recent PHP 7.0 from the end of year 2015.
- Windows version of PHP uses PRNG from Visual C++ stdlib library,which is linear congruential generator with 32-bit internal state and 32-bit output - output is the same as internal state, so the generator has no hidden state.
- Linux version uses glibc implementation of rand(). This is also linear congruential generator, but in this case with much larger 1024-bit space (32 32-bit integers). Native output of the generator is 31-bit number (last bit is cut of).

1.6.5 mt_rand()

Most popular PRNG for PHP. This generator has known and consistent characteristics regardless of PHP version or host operating system[16]. The source code of this function mentions that this is MT19937 implementation of Mersenne Twister, but I have found that mt_rand() differs a little and produces different output than MT19937 (this will be discussed in later chapters). This function has an internal state of 624 integer values (19968 bits) and very large period of 2^{19937}-1. However, this function does not offer cryptographically secure output, making this function ideal target for my toolkit.

1.7 PRNG Cracking

If pseudo random number generator is supposed to provide seemingly random values, process of PRNG cracking is trying to make this values as "un-random" as possible, sometimes even not random at all. By cracking and predicting PRNG, an attacker has a lot to gain since PRNG outputs are often very valuable:

- Password recovery links in recovery emails

- System generated user passwords
- Lottery numbers
- Encryption keys (HTTPS, TLS, SSL...)
- Hidden identifiers
- Access tokens

Predicting outputs of PRNG is not the only exploit arising from poor PNRG usage. Seed recovery can leak critical information to the attacker, since some applications use secret informations as PRNG seeds. Some cracking methods can reveal how many times was the PRNG used by the application - usually not providing critical info to the attacker, but should still be considered as possible vulnerability.

1.7.1 Basics of PRNG cracking

In order of the attacker to know the future outputs of the PRNG, he must know two informations about the target application - type of PRNG used and what is the current state of the PRNG. The first information is rarely a problem for attacker - as previously discussed, only handful of PRNGs are used in reality, the PRNG used for each output is the same and source code for the target application is often accessible, so it easily be assumed the attacker knows or can trivially guess this information.

1.7.2 State recovery

Since PRNG output is entirely based on the PRNG used and the PRNG state, the attacker must also know the current generator state. Good and secure PRNG will never leak the current state. Sometimes, it's not required to find the state directly - the attacker just needs to find the seed that was used to initialize the generator and this information can be easily used to reconstruct the current state just by replicating the PRNG behavior on the attackers side.

1.8 Cracking PRNGs in PHP

In this section, I will describe issues with cracking PHP PRNGs and existing algorithms used for cracking various rand() implementations and mt_rand() in PHP.

1.8.1 PRNG output truncating issue

Both mt_rand() and rand() have some ”native” output size. For mt_rand(), the generator outputs 31-bit number, for rand(), it can be 16, 31 or 32 bits, depending on platform and PHP version. But sometimes, a programmer needs much smaller output, for example a number between 1 and 10. In this case, both rand() and mt_rand() offer bounding of the output between two numbers. If the user requests a number between 0 and 255 (8-bits), PRNG like mt_rand() needs to scale the 32-bit output to this range and loses 24 bits of data in the process. This means the attacker can extract much less state data from each generator output.

1.8.2 Seeding time issue and Apache

As mentioned previously, both rand() and mt_rand() require to be seeded before use. However, developers don’t need to do this manually (srand()/mt_srand()) and they rarely do so – PHP does this automatically but the question is when and how.
In order to answer this, one thing must be explained. As of May 2017, most PHP servers currently use Apache to handle user requests.[17]
Each user request is handled by some Apache process which is already running. If the the PHP script that the user is requesting uses PRNG, this PRNG is seeded at the first call. However, if another user comes to the same Apache process, he will access the same PRNG as the user before him and this PRNG was already seeded, so no need to reseed it. This brings a new issue - there may be many PRNG outputs between seeding and an attacker getting PRNG output, which is bad for seed brute-forcing, as will be explained later. Ideally, an attacker wants PRNG that was freshly seeded, and luckily there are methods to force Apache to reseed its PRNG.
One could ask what value does PHP use for seeding. A lot of programmers use trivial seeding values, such as current time. This is terrible solution, since current time is absolutely not unpredictable. Very old versions of PHP used entropy from multiple sources, which could lead to some seed attacks, but it was fixed long time ago and majority of PHP servers deployed use /dev/random, which is cryptographically secure PRNG[18].

1.8.3 Seed brute-forcing

Seed bruteforcing is probably the easiest and most straightforward way of cracking PRNGs. Let’s have some PRNG (doesnt’t matter if rand() or mr_rand()), seed it with some hidden value and then output 5 public numbers. Since both generators are seeded using 32-bit number, there are only 2^{32} ways of seeding this PRNG.
An attacker could simple try all 4,294,967,296 numbers to seed the PRNG, output 5 numbers and if the 5 numbers match, he has revealed what was the

hidden seed value. For this attack, he needs to output 5*2^{32} numbers from PRNG, which is around 21 billion outputs, a number of computations possible to compute on a single machine in matter of hours.
However, in this example it was assumed, that the attacker got his 5 numbers immediately after seeding. As mentioned previously, this might be hard, because the attacker will most likely receive received outputs from PRNG which was seeded a long time ago and might have produced thousands to millions of outputs. He needs to check much more outputs per seed and the computation suddenly becomes unfeasible.
Even though this method has many issues, this thesis will continue with main focus on this method, since it is the best method for real-world attacks.

1.8.4 Cracking rand() - Windows

Cracking rand() function running in Windows environment is actually really easy. Since state size of Windows implementation of rand() is only 32-bits, we can just trivially try all the possible states and check, if some state is returning the same sequence as the target server. It should take only few minutes, if parallelization and vector computations are used, times under one minute should easily be possible.[19]

1.8.5 Cracking rand() - Linux

Since most PHP servers run Linux, this version is more used than the Windows version. In Linux environment, rand() uses glibc implementation of rand(). This means, that the state size is 1024, which is too many possible states to brute-force. So the only possible methods left are brute-forcing or some advanced state recovery attack. There are some state recovery attacks possible by solving complex systems of linear equations, however they are using complicated math and are far beyond the scope of this thesis, so I will focus on seed recovery methods.[18]

1.8.6 Cracking mt_rand() – State recovery

State recovery attacks for mt_rand() are also possible - these attacks don't aim to recover the seed, so they can be used also for PRNGs, that have been seeded a long time ago. However, for this attack, the attacker needs to get 624 full consecutive outputs from the PRNG, since mt_rand() is returning output from different state integer on each call. Getting 624 consecutive outputs is actually really hard in reality.
Another issue is output truncating discussed previously – this can lead to necessity to output multiple of 624 outputs and then solve systems of complicated linear equations. Some cracking methods like this for mt_rand() have been proposed, but in reality they are hard to implement, in most cases impossible to execute, so they will not be implemented in this toolkit.

1.9 Existing software for PRNG cracking

Cracking sequences generated by PRNGs is nothing new and many attempts to create a PRNG cracker have been already made. These are some of them for mt_rand() cracking.

1.9.1 php_mt_seed

A tool that was originally made public on openwall.com as one of the first mt_rand() crackers[20]. It uses seed bruteforcing technique to recover the seed of a given number sequence. It supports cracking numbers from a specified range, but no advanced parsing options or cracking techniques. It's main strength is very high optimization – it can leverage AVX2 vector instructions, which helps greatly when running on modern CPUs and even more on Xeon Phi accelerators. This is probably the fastest solution for seed bruteforcing available.

1.9.2 Untwister

Another tool for PRNG cracking is Untwister[21]. This tool supports multiple PRNGs based on Mersenne Twister such as ones used in Java, Ruby, glibc and PHP. It supports seed bruteforcing, but at much lower speeds than php_mt_seed, making this option impractical as will be shown in Testing chapter. However, this tool can use state recovery methods discussed previously, which other tools don't support. It also has more advanced user interface with support for using files as inputs.

Chapter 2

Design

The second part of this thesis will explain what is expected from the final product and how is it going to be implemented. Focus is going to be given on application requirements, use cases, architecture and other aspects of software engineering.

2.1 Basic explanation of toolkit

In simple terms, the main goal of the toolkit is finding and exploiting PRNG vulnerabilities in PHP applications. The core functionality of the toolkit should allow the user to find the seed of the given number sequence. In order for the toolkit to be useful in practice, it will also include few tools to support this whole process. This will include PHP PRNG emulator, tools for connecting to Apache process and few more.

2.2 Target users

As previously mentioned, it is expected that the core user group of this toolkit are going to be web application penetration testers. These users are expected to be natural with command-line based tools and basic scripting with Python. However, it cannot be expected from these users to know details about PRNG cracking, so the toolkit must be reasonable simple to use.

2.3 Non-functional requirements

- **N1 Multi-platform** : Application is required to run on multiple different operating systems, with main focus on Windows and Linux. Although it is expected that most users will run Linux (Debian-like systems like Kali), some users prefer using Windows environment.

- **N2 Responsiveness** : Application needs to run quickly and quickly respond to user commands.
- **N3 Easy-to-use** : Application should be accessible even to users which don't know the specifics of PRNG cracking.
- **N4 Streamlined workflow** : Application must be designed in a way which will allow the user to use it in professional environment without slowing down the user workflow.

2.4 Functional requirements

- **F1 Open and hold connection to server** : Function to force target server to create new Apache process.
- **F2 Save data from server connection** : Tied to F1, function required to capture data from fresh Apache process.
- **F3 Define input formatting** : This is required if the target application uses random numbers from some sequence.
- **F4 Parse input sequence : based on formatting** : Application needs to be able to parse input to number sequence from previous formatting command.
- **F4 Crack sequence** : Core functionality of application, find the seed of the target number sequence.
- **F5 Seed PRNG** : Create a new PRNG in memory from the supplied seed.
- **F6 Get PRNG output** : Get output from selected or all in-memory PRNGs based on formatting from previous commands.
- **F6 Find PRNG output** : Find which PRNG outputted the requested number from the previously seed PRNGs.
- **F7 Seek PRNG** : Move forward in PRNG period.

2.5 Use cases

In this section, several use cases are going to be listed, which describe how the application is going to be used and interaction between the user and the application.

2.5.1 U1 – Get data from fresh PHP processes

Let’s have a remote PHP application running on Apache server, which generates number sequences using PRNG. In order to get ”fresh” PRNG outputs, we need to get fresh Apache process (details of design will be discussed later).

1. User enters target URL to the toolkit alongside other input parameters such as number of connections to create, protocol, etc.
2. Toolkit then creates first set of open connections to the target server and keeps them open.
3. After a while, toolkit will open second, smaller set of connections, forcing the target server to spawn new Apache processes.
4. Toolkit will close the first set of connections, forcing Apache to kill processes hosting these connections.
5. Toolkit will download and save data from second set of processes and also close the connections. Processes hosting these connections will probably stay alive.

2.5.2 U2 – Crack PRNG sequence

In this case, the user has already a sequence he wants to crack (for example obtained from the first use case) and he wants to find the seed of the sequence.

1. (Optional) User enters formatting sequence to supply formatting metadata for the cracker.
2. User enters a sequence to be cracked.
3. Cracker tries to crack the sequence and if successful, outputs possible seeds to the console.

2.5.3 U3 – explorer further PRNG outputs

Let’s say there is a webserver with 10 processes running on it and the user knows the internal state of these processes. User also has another output from the application, but he needs to know which PRNG generated this sequence. The application should be able to find this and also simulate more outputs.

1. Users initializes PRNG in memory using seeds from previous crack command.
2. User enters PRNG output to be searched for.
3. Toolkit finds which PRNG generated the output and how many outputs have the PRNG generated in between.

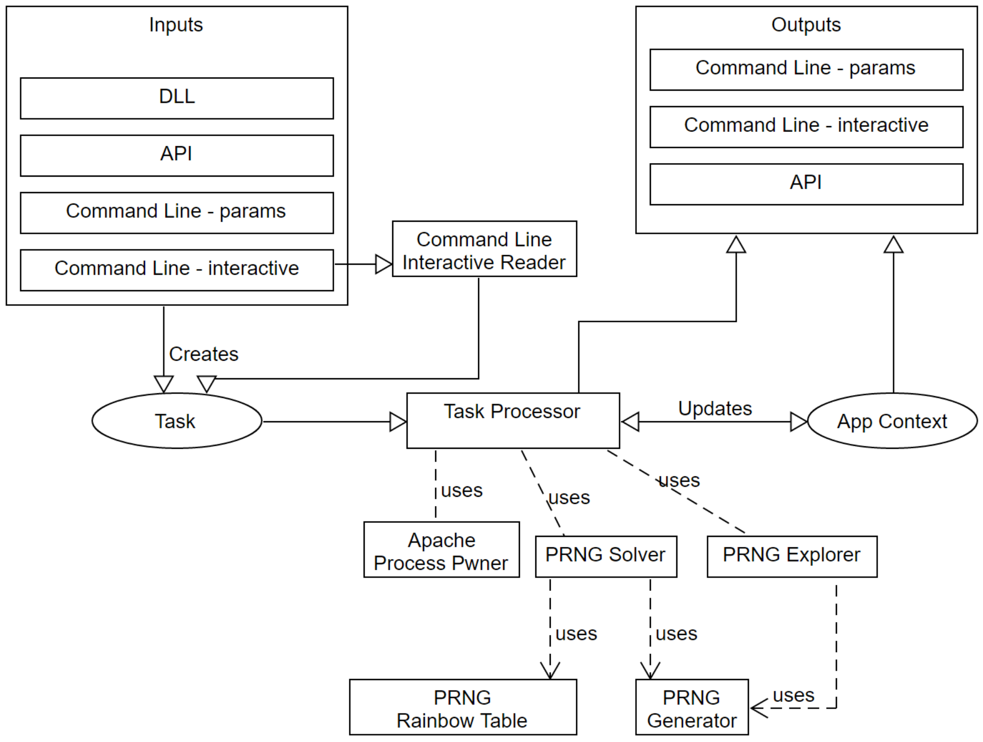


Figure 2.1: Application architecture

4. (Optional) User can continue to explore what outputs has the generator outputted before or what is going to be generated later.

2.6 Application architecture and components

The application is divided into two parts, where the first part is responsible for input handling and Task creation and the second part processes the Tasks. Task is an universal object for storing information about what the application should do.
A general architecture description can be seen in Figure 2.1. In interactive mode, all inputs are handled by Console Input Reader which produces Tasks. These tasks are processed by Task Processor, that decides which component should execute the task based on the Task type.
Although the application component diagram mentions multiple inputs for task creation, this thesis will focus only on CLI interface. More input/output interfaces could be easily implemented, but they are beyond the scope of this thesis.

2.6.1 Apache Process Pwner

Component used to connect to the target Apache web-server. This component will be used to handle connections, download data and save results to local hard-drive.

2.6.2 Task Processor

This component will receive task from any input reader in universal format and try to execute the task. Execution of task consists mainly of routing it to the right component, which is able to execute it.

2.6.3 Console Input Reader

This component will handle raw user interaction with interactive command-line interface. When the user enters the task in CLI, this component will parse the task and when successful, it will send the task to Task Processor component.

2.6.4 PRNG Cracker

This component receives task as a series of numbers to crack. After the algorithm searches for all possible seeds, it will output the possible seeds for this series.

2.6.5 Rainbow Table Reader

This component is used by PRNG Cracker component and it is used to access the rainbow table files and provide rainbow table cracking services. This component will be discussed further in the Realization chapter, since it was initially not intended to be included in the toolkit. However, the performance improvement was so large, I decided to implement it.

2.6.6 PRNG_PHP

Emulator of PHP mt_rand() PRNG re-implemented in .NET Framework. Provides faster execution than PHP variant and is used to emulate PHP PRNG by PRNG Cracker.

2.6.7 PRNG Explorer and generator

This component will be used once the user finds the desired seed. For example, let's say the target Apache/PHP server is running 12 processes and an attacker has found seed of all this PRNGs. He then asks the server to send a password recovery email for some victim user. This email is going to contain randomly generated recovery token, which the attacker wants to guess.

However, the attacker doesn't know, which process the user connected to and which PRNG was actually used.
In this case, the attacker can request the server to output each PRNG (using the same method as getting the initial PRNG output). This output is then entered into the PRNG explorer. The explorer will output few outputs from each generator and find out not only which PRNG matched the output, but also how many times the PRNG was used between initial seeding and this search. The PRNG that outputs something is the one that was used to send the email to the victim user.
The generator can then reveal future outputs, which are going to contain the desired recovery token. This attack will be simulated in the Testing chapter.

2.7 Mode of interaction

For this application, I have chosen command-line interface (CLI) . This should be not a problem but actually an advantage, since most users are accustomed to CLI anyways. Since PRNG cracking is usually multi-step process as it can be seen in Use-Cases, an interactive CLI was chosen. Toolkit will have some context between commands which can be used for command input. Commands will consist of two parts, command name and optionally parameters.
Considering the functional requirements of the application, these commands will be available to the user :

- `overload` *`target_url`* [-n1,-n2,-o]
  Opens a set of connections to the target server. After a while, the application will open second smaller set, which is a set of connections, whose host processes should remain alive. After a while, first set of connections is close and later the second set of connections is closed and their output saved to disk.
  `-n1` *`n`* : Number of connections in first set, default is 15
  `-n2` *`n`* : Number of connections in second set, default is 15
  `-o` *`filename`* : Filename prefix of output files, default is `connectionData`

- `crack` *`element1 element2 ... elementN`*
  Core functionality of the cracker. Searches for a seed for a given sequence. The number of the elements must be the same as speified in the `format` command. Each element is delimited by a space character. After the operation finishes, it outputs possible seeds and execution time in milliseconds to the console.

- `format` *`format1 format2 ... formatN`*
  Sets the formatting meta-data for the cracker, generator and other components. Formating types are listed in separately in section 2.8.2.

- `seed` *`n`*
  Seeds a new instance of mt_rand() PRNG in memory for later use.
  `n` : Number to use as a seed. Must be able to fit in 32-bit unsigned integer.

- `seedlast`
  The same as `seed` command, but uses the output from last cracking command as input. This command is present mainly for faster workflow.

- `loadcrack` *`filename`*
  The same as `crack` command, but instead loads the inputs from the supplied file. This can be used to execute a number of cracking tasks in sequence, since each each input of the file is treated as separate crack task.
  `filename` : Name of the file to use as input.

- `find` *`element1 element2 ... elementN`*
  Searches all the initialized PRNGs and checks, if any on them can produce the selected sequence. If some generator satisfies this, the command prints the generator ID, how many states the generator produced since seeding and the original seed. Uses the formatting from the `format` command.

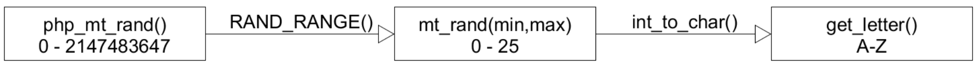
- `loadfind` *`filename`*
  The same as find, but uses file on disk as input.

- `resetformat`
  Resets the format to the default format (one native 31-bit output) .

- `generate`
  Generates the output from all the initialized PRNGs based on the current format.

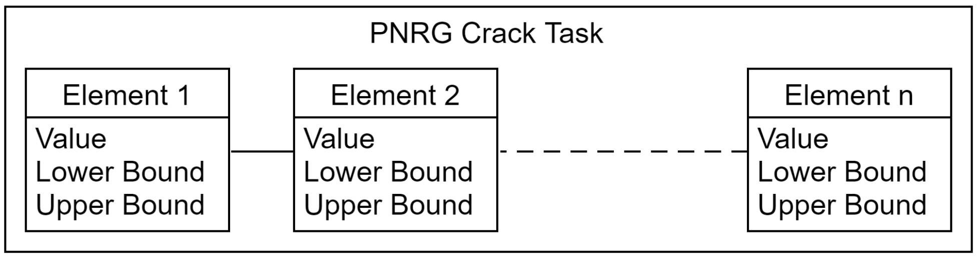
2.8 PRNG Cracker Design

Every PRNG is implemented in a way that produces series of numbers in their native form. Since programmers rarely need numbers in this form, they often use function, which generates random numbers from some range. Then they can convert numbers from this range to some other symbols, for example alphabet letters.



The PRNG Cracker is guessing seed from the PRNG outputs, so the cracker

needs to get this native output back. First, the cracker reverses the last transformation. This is not trivial, since the transformation can have many forms - the alphabet could be reversed, meaning number 0 is transformed to 'Z' and not 'A'. Luckily, there are not so many transformations and they are fully reversible.
The second transformations is a little problematic. Since the transformation is transforming from a large range to a much smaller range, a lot of data is lost in the process. There is no way to recover this data, so there are many possible outputs that can be transformed to the same number. The smaller the target range, the larger the number of possible outputs, in this case it's $2^{31}/26$ possible outputs that transform to the same number.
In order for the cracker to be as flexible as possible, it just needs to know the values generated by mt_rand() and from what range they were generated.



With this information the generator can do the actual cracking. It goes through all possible seeds (seed bruteforcing mentioned previously) and if some seed generates the target sequence (this is the reason the cracker needs to re-implement mt_rand()) , this seed is returned as a result. If the sequence is very short, there can be more seeds that satisfy this, so all of them must be returned.

2.8.1 PRNG Crack Task Parser

As mentioned previously, PRNG Cracker is cracking a sequence of outputs from mt_rand(). However a user would like to enter a direct output from the target application. This could be anything, from Universally unique identifier to a hex number.
This parser consists of two modes, blind and formatted. In blind format, the user will not enter any formatting information about the input, so the parser will have to guess this information using some basic heuristics.
In second formatted mode, the user first describes the sequence he wants to crack, e.g. he knows that it's a sequence of 5 words, each word consisting of 4 hex letters and each word generated by single mt_rand() call. The formatter then knows that this can be transformed to sequence of 5 integers with values between 0 and 65535 (2^{4*4}).

2.8.2 Sequence formats

The user has an option to specify what elements is the target sequence going to contain. Each element is going to be delimited by a space character. Each element is specified by text identifier and it can be followed by a specified range following a comma. Generaly, each element will look like this :

$$identifier[, min - max]$$

Few formats have been proposed, but much more could easily be added. For now, only the decimal identifier is supported with range definition.

- n : Identifier for decimal number. For example a decimal number between 0 and 1000 will be specified like `n,0-1000`.
- h4 : Four character hex identifier. Translates to a decimal number between 0 and 2^{16}-1. Alphabetical symbols can be both lower and upper case. Example : `30FF` is parsed to 12543.
- h2 : Same as h4, two he characters, range is 0-2^{8}-1
- h8 : Same as h4, two he characters, range is 0-2^{32}-1
- uuid1 : Universally unique identifier. Although UUID should be standardized, in reality many different implementations are used and some of them use only output from mt_rand(). The following implementation is grabbed from Adobe Developer portal as "good" implementation of UUID.[22]

```
function UUIDv4()
{
return sprintf(
    '%04x%04x-%04x-%03x4-%04x-%04x%04x%04x',mt_rand(0,
    65535), mt_rand(0, 65535),mt_rand(0,
    65535),mt_rand(0,
    4095),bindec(substr_replace(sprintf('%016b',
    mt_rand(0, 65535)), '01', 6, 2)),mt_rand(0,
    65535), mt_rand(0, 65535), mt_rand(0, 65535) );
}
```

  This will be parsed to a sequence of integers, most of them in range between 0-65535 (2^{16}-1). This format is used surprisingly often.

To show an example, let's say a user wants to find a seed of this sequence : a random number betwen 0 and 100000 and followed by uuid specified above. He would enter a formatting command like this:

```
format n,0-100000 uuid1
```

And then enter the cracking command:

```
crack 74030 8f7e289f-c8fb-4734-8907-7595f625acba
```

That sequence was generated using the following PHP code :

```
<?php
echo mt_rand(0,100000)." ".UUIDv4();
?>
```

2.9 Apache Process Pwner design

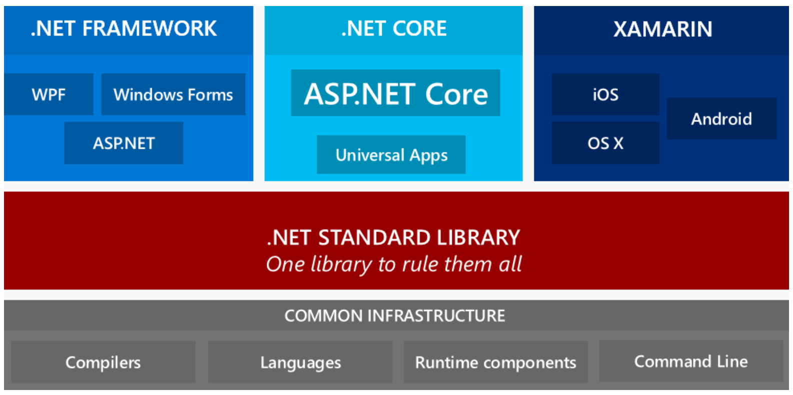
This component of the toolkit is responsible for forcing Apache to spawn new processes with freshly seeded PRNG. This is a form of attack based on method proposed by George Argyros and Aggelos Kiayias.[18]
As mentioned in previous chapters, if PHP is running under Apache, each user is accessing only one process at a time. Default installation of Apache is keeping around 10-12 processes alive and ready for incoming users, so the server can service up to 12 users without creating new processes, which is CPU expensive operation. This doesn't sound like a lot, but each process can service new user as soon as the previous user is finished downloading the request and 10-12 processes is actually plenty for most of the websites.
However, the user has an option to connect to more than one process at a time. In order to do this, the user of the server needs to open few parallel connections to the server. Each new connection connects to a different process and if the number of connections is larger than number of unoccupied processes, Apache is forced to spawn new processes (with freshly seeded PRNG !). By performing few tests, it was discovered that default installation of Apache doesn't limit a single user in number of open connections. Even if the server would be configured in a way that limits the number of open connections per user, an attacker could use a fairly small network of proxy servers (this will not be implemented in this thesis, but should be easy to add later).
Once the Apache sees that it has a lot of unused processes, it starts to shut them down, or until it is at the default value of 12 processes. The first processes to be killed are the ones that have been unused for the longest time. Apache Process Pwner (APP) makes sure the newly created processes stay alive - it does it by keeping the connection to them open as long as possible (how it does that is discussed in Realization chapter). APP also saves response from this processes, which can contain sequence data for PRNG cracker and this sequence data is going to come from freshly seeded PRNG.

CHAPTER 3

Realization

3.1 Choosing a development platform

As mentioned previously by our non-functional requirements, application should be multi-platform and high-performance. For this task, many solutions are already available. Python looks like and obvious choice, however, in many tasks it's performance would be a serious bottleneck.
Another choice is Java, which is fast enough for this toolkit and portable to more platforms than required, but personally I don't prefer Java Programming language and there is a better option available.
I have been using .NET for professional projects for a few years now and I have grown very fond of it. At first, I was afraid of it's performance compared to compiled native languages like C/C++, but after testing few PRNG generators coded in both C++ and C#/.NET, there was almost no performance difference.
Another concern was portability – this is nicely solved by the newest .NET Core branch of .NET. It builds on the .NET Standard and is available on Windows, Linux, Mac and in Docker environment, fully satisfying non-functional requirements.



3.2 Implementing mt_rand() in C#

As discussed previously, C#/.NET version of PHP's mt_rand() was required to be reimplemented. This component is used to verify results and check which seeds actually generate the requested results. PHP function mt_rand() is based on Mersenne Twister. The source code for the function mentions MT19937[16], so it can be assumed that mt_rand() uses standardized MT19937 implementation of Mersenne Twister.
After implementing standard MT19937 generator in C#, it was discovered that this generator was outputting different number sequences than mt_rand(), so reverse engineering the PHP source code was required.
From the source analysis, it can be seen that mt_rand() implementation is fairly simple. The generator has an array of 624 32-bit integers that hold the current state of the generator. The generator must first be seeded – this is done at first call of mt_rand() or it can be done manually by calling mt_srand(). Internally, mt_srand() calls php_mt_reload() function, which is doing the actual "twisting" in the algorithm.

```
uint32_t *p = state;
N = 624;M = 397;
for (i = N - M; i--; ++p)
        *p = twist(p[M], p[0], p[1]);
for (i = M; --i; ++p)
        *p = twist(p[M-N], p[0], p[1]);
        *p = twist(p[M-N], p[0], state[0]);
```

Listing 3.1: php_mt_reload()

As it can be seen, the algorithms takes pairs of numbers in the state and mixes them up to produce new numbers.
After the state is initialized, mt_rand() takes one number from the state and outputs it on each call of mt_rand(). This means that after 624 calls, the state must be "retwisted" and php_mt_reload() is called again. Each mt_rand() call looks like this:

```
s1 = *state(next)++;
s1 ^= (s1 >> 11);
s1 ^= (s1 <<  7) & 0x9d2c5680U;
s1 ^= (s1 << 15) & 0xefc60000U;
s1 ^=  (s1 >> 18);
return s1>>1;
```

Listing 3.2: Core of mt_rand()

As seen above, mt_rand() cuts of the last bit of the output on the last line, which is something not included in standartd MT19937. This means that PHP implementation of MT19937 outputs only 31-bit numbers ! This has some serious consequences.

First one is that this generator in incompatible with other MT19937 generators. Although generally not an issue, it took me few hours of debugging, since I assumed PHP uses correct MT19937 implementation and I was really surprised when my reimplementation generated different sequences.
The other more important issue is that all Mersenne Twister cracking methods assume that on each mt_rand() call, the generator outputs full 32-bits. If the bit-shifting operation is skipped, the whole algorithm can be fully reversed – if the output of the generator is captured, one integer of the state be recovered. If 624 consecutive outputs are captured, the whole state can be trivially recovered. This is not possible in the PHP implementation of MT19937 – the last bit is always lost and there is is no way to recover it. Many theoretical attacks on mt_rand() don't consider this fact and thus they are not actually possible in reality.
Seed bruteforcing attack is still possible in the same way.

3.2.1 Changes in PHP 7.1.0

During the development of this thesis, PHP team released a new version of PHP, which makes important changes to mt_rand(). Since version 7.1.0 mt_rand() uses correct MT19937 implementation, without any bit-shifting, so previous attack on mt_rand() is actually even simpler, since the state can be easily captured.
This issue was reported only recently in 2016 as bug #71152 and fix was submitted the same year.
Since most PHP environments don't use PHP 7.1 yet, since it was released only recently, I will be targeting only the older versions mt_rand(). Although the implantation is now fixed, with some very minor changes to my cracker the toolkit works also on newer versions of mt_rand() and some previously discussed attacks are actually easier. However, all the currently existing PRNG cracking tools are not yet updated for this new version.

3.3 Implementing seed bruteforcing cracker

This is the actual main component of the toolkit. The component is very basic in the core concept – it loops through all possible seeds, generates the sequence according to the input format and if the output matches with the target sequence, this seed is added to the list of outputs.
This algorithm was implemented to use multiple threads to speed up searching for the seeds. It was discovered that this scales directly with number of threads with no performance hit from synchronization, since there is no synchronization required – each thread is assigned the same amount of work and it is expected to finish at roughly the same time – this is not exact, so in the last few moments of cracking, not all CPU cores are going to be used. However, this last phase is usually only few seconds long and it not worth to

implement more advanced thread synchronization.

3.4 Rainbow tables for seed finding

During the development of this toolkit, I came up with an idea of implementing rainbow tables for seed recovery. It was an idea I came up with while watching some security conference. I started searching for some mentions of it, but not only nobody tried to implement it before, Google was unable to find a single mention of this method. This was the reason I dismissed the idea at first.
But later I tried a fast and simple proof-of-concept application, and it showed the method might by viable, so I decided to explore this further and soon it became the main feature of this application. Since the architecture of the toolkit was designed for high flexibility, it was very easy to include this component so late in development.
Rainbow tables are nothing new in cracking and security applications. They are used mostly for reversing hash functions. They do this by first computing all the results (e.g. hashes of popular password), sorting them and then instead of hashing all passwords for each hash reverse, the user can just look up the hash in the results directly (since they are sorted, this is instant) and easily recover the password from the hash. It is the typical example of space/-time trade-off.
Next section is going to show how can be this method used with seed recovery attacks.

3.5 Implementing rainbow table generator

The rainbow table for this toolkit is going to be used to lookup seeds of PRNG outputs. However, there is one problem – multiple PRNG outputs can have the same seed. PHP mt_rand() is seeded by 32-bit number and produces 31-bit output(the last bit of output from mt_rand() is discarded). This means, that there are 4,294,967,296 possible seeds and 2,147,483,648 possible outputs (in this context, output means first number that PRNG generates), which guarantees that there must seeds with the same output, so this is not one-to-one relationship.
At first, it would be useful to know, how many "multiplicities" there are - PRNG outputs with same seed.
In the first test, I have created a program which creates an array of bytes for each output, setting the initial value to 0. Then the program goes through all 4,294,967,296 possible seeds and gets the first output of PRNG for each seed and then increments the value in the array using the PRNG output as

index. After this program finishes, at the n-nth position in the array should be number of seeds that generate output n.
After this program finished, I created a script that would count how many PRNG outputs had 0 seeds, 1 seed, 2 seeds, etc. and this was the result:

# Seeds	Count
0	290634907
1	581260248
2	581239311
3	387541467
4	193731467
5	77501756
6	25838524
7	7381257
8	1844208
9	410505
10	82132
11	14978
12	2473
13	348
14	59
15	7
16	1
17	0

As it can be seen, there are 16 seeds that have the same number as their first output – let's call this first output "image", which is number 423,148,348 the only number with this interesting property.
This caused a little problem during the development since in one of the early iterations of the toolkit, each PRNG output had only 4-bit counter for number of seeds, since I wrongly assumed there are no numbers with this property and the program was silently overflowing causing a range of problems.
It can be also seen that 290,634,907 outputs have no seed – there is no way a PHP mt_rand() is going to output these numbers as first output, so an average output has 2.31 seeds.
So, after it is known how many seeds each output has, the rainbow table can be constructed. It should work like this – the target PRNG output for which the application is finding the seed is used as index in the rainbow table. But this is not that straightforward, since some outputs have multiple seeds and some have none, so the algorithm cannot index the seed table directly.
Index table is required first, which will tell contain the target index in the final table. The value at index `n` is going to be the position in the second table for the output `n`. Generating the first "index" table is easy, the table generator

just uses the first table with seed counts and iterates through it, each time increasing the current index by the number of seeds for given output.

```
long positionCounter = 0;
for (long i = 0; i < (UInt32.MaxValue+1L)/2L; i++)
{
  int numSeeds = countsFile.ReadByte();
  indexFile.Write((UInt32)positionCounter);
  positionCounter += numSeeds;
}
```

Listing 3.3: code for generating index table

This table will take 8 GB of memory, since it stores 32-bit positions for 2^{31} images.
Now, the generator creates the final seed table, which is going to contain the seeds for each PRNG output. This table is going to be the largest : 2^{32} 32-bit seeds, or 16 GB. The generator first populates the table with zeros and slowly fills it with seeds :

```
long numSeeds = UInt32.MaxValue + 1l;
long numImages = (UInt32.MaxValue + 1L) / 2L;
uint[numSeeds] preimageBuffer;//this contains only zeroes
   now
uint[numImages] indexTableBuffer;//this contains data
   from index table
for (long i = 0; i < numSeeds; i++)
{
  var seed = i;
  var image = PHP_MersenneTwister.GetFirst((UInt32)seed);
  var preimageTablePosition = indexTableBuffer[image];
  var existingValue =
     preimageBuffer[preimageTablePosition];
  while (existingValue != 0);
      preimageTablePosition++;
  preimageBuffer[preimageTablePosition] = seed;
}
```

Listing 3.4: code for generating preimage table

Since the table generator is accessing this table and also indexing table very randomly, HDD or even SSD would be way too slow for this, so they need to be stored in RAM. This makes a requirement of at least 24+ GB RAM for target computer to run this. The memory was needed to be split to 64 KB blocks, since .NET(and many other SW platforms) doesn't support memory blocks of this size.
However, this table needs to be generated only once and users don't have to generate it at all, they can just download it from the Internet instead.

3.5.1 Faster mt_rand() for rainbow table generator

Generating the rainbow tables was a very slow task so I was trying to find any way to make the process faster. I soon realized that the process seeds the whole Mersenne Twister, initializes it and then gets only one output from it – the rest of the state is never used. This means that the runtime needs to allocate and initialize 19968 bits of memory for each table entry.
I decided to create very lightweight version of mt_rand(), which would be able to generate only one output – exactly how much it is required for rainbow table generation. Since how mt_reload() is implemented, this generator requires only 2 state values and this can easily fit into CPU cache, making the whole process much faster.
Some very quick testing showed that after switching to this faster generator, CPU time spent on generating numbers decreased 20-times.

3.5.2 Running the table generator

Since I needed to test this functionality, I needed to obtain powerful enough machine. First few iterations of table generator required 32GB of memory, since they were not optimized enough. The only way for me I could run this program was to launch an Amazon EC2 Instance with 60 GB RAM and 8 CPU cores. I was able to easily parallelize the code – I have ignored any synchronization problems, since the possibility of two threads generating two seeds with the random number output at the same time was extremely low and this was confirmed during testing. The computation of all the tables took only 2-3 hours, as opposed to 10-15 hours without parallelization.
One hour of Amazon EC2 r3.2xlarge instance running Windows Server in Frankfurt region costs about 1.2 USD per hour, so this computation was rather cheap and could be made even cheaper using ”spot” instances (computation usually during night, when Amazon has a lot of spare capacity).[23]

3.6 Implementing the rainbow table cracker

Since I have generated the rainbow tables, finding the seed of PRNG output is fairly simple. If the target PRNG output is known, full list of seeds can be retrieved almost instantly from the rainbow table. However, this works only if the cracker has full 31-bit output of PRNG. The problem arises when the PHP application needs random numbers from a specified range and PRNG tosses away a part of the output, for example when generating a random number between 0 and 65536.
Here is again the problem with transforming native mt_rand() output, the programmer requested number from 0-65535 range, while native output from PHP mt_rand() is from 0-2147483647 range. Let's say, there are three different seeds, and the first outputs of PRNGs from these seeds are 1000000,1000001

and 1000002. However, if the PRNG would be asked to generate numbers from the reduced range of 0-65535, all three outputs would be transformed to the same number : 30. So although the attacker could know the output of the target PRNG was for example 4670, he cannot know what was its real output, since 15 bits of data were lost in the process.
However, the cracker can just try all the possible outputs. If the cracker scales back the number 4670, he discovers that the native output of the PRNG was somewhere between 153026560 and 153059328 – that's 32768 possible outputs. To crack 32768 outputs using conventional brute-force state-of-the-art brute-force method would take one year even while using powerful GPU. However, the rainbow-table cracker can do the same task in milliseconds.
Once the cracker finds the potential seed to the first output, it will confirm the seed against all the other outputs in the target sequence. If the rest of the output matches, the sequence was successfully cracked.

3.6.1 Downsides of rainbow-tables

This method works very well for cracking numbers from larger range. Obviously this method doesn't work if the cracker doesn't have the first PRNG output available, since the first number is required for rainbow-table lookup (however, second number rainbow-tables are possible to be generated, but probably too complicated for real use) .
Another issue is cracking numbers generated from very small range. If the cracker has a series of hex symbols to crack, where each hex symbol was generated separately, it needs to check seeds for 134,217,728 PRNG outputs ($2^{31}/16$). Although this looks like scary large number, even this can be cracked. In this case, the rainbow tables could be loaded to RAM and make the whole look-up process much faster. This computation will take longer time, but it is still in pretty reasonable time – more will be discussed in ”Testing” chapter. The major downside to this is that user will need 24GB of free RAM to use this functionality.

3.7 User input parsing

Personally I have already implemented several CLI-type tools before, so I decided not to be fancy and just create very simple interface. Each user command starts on a new line and is defined by a keyword. All parameters come after the command keyword. If the keyword is not recognized, the program executes the help command. Output of the command parser is instance of ”PRNG Task”, which is then handled by the Task Processor.

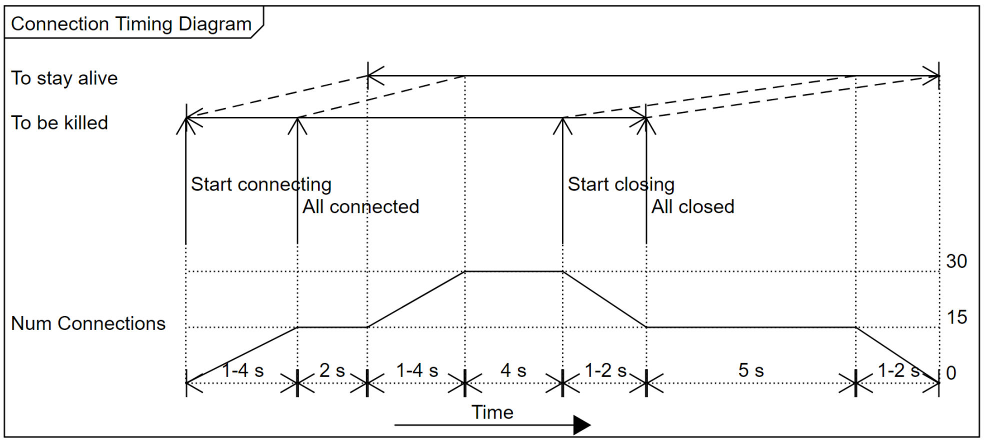


Figure 3.1: Apache Process Pwner timeline

3.8 Apache Process Pwner implementation

After trying several different different implementations of the APP, the following attacks have been implemented:

1. Open first set of connections to occupy all existing Apache processes.
2. Open second set of connections to force Apache to spawn more processes.
3. Close first set of connections.
4. Download response from second set of connections and close them.

The timeline of the whole process can be seen in figure 3.1. During the implementation, a new problem was discovered – first set of connections must stay alive until at least all of second set of connections are open. If this doesn't happen, some connection from the second set can be handled by an existing process, which will fail to spawn new process.
An obvious solution to this was to open connection as fast as possible, possibly by making conenctions from server that is nearby. However a much nicer and more reliable solution exists by manipulating speed of request downloading.

3.8.1 Slow loris attack implementation

Connections can be forced to stay open longer by manipulating the speed of handling network requests. Instead of using basic network components to handle requests and responses, slow loris attack was implemented.
This attack was invented by Robert Hansen[24]. The attack sends and downloads requests byte by byte and waits a little bit between each byte. This way the server is forced to let the connection be open for very long time without time-outing, since the user is actually active, but at very slow speed.

Even a simple GET request with no parameters is usualy pretty big. A simple GET request from Chrome web-browser to localhost server with no data or parameters has 415 characters, so we can expect every request from attacker to contain at least 200 characters. If the attacker wants to keep the connection open for 10 seconds, he just waits for 50 ms between each character sent. After 200 charaters (or 10 seconds), he can choose to close the socket or upload and read response at the normal speed if the response is required.
Response can be required, since this is the first response of the given process. If the GET request contains request for page with random sequence, the attacker will want to save this request and extract the sequence later. This could be done using wide range of tools, for example using built in text editing tools in bash such as cat, top, cut, awk or sed.

Chapter **4**

Testing

In this chapter I will check the toolkit functionality against few vulnerable applications. The applications are just proof-of-concepts created just to showcase the toolkit.

4.1 Testing configuration

For these tests only my laptop is used. Host operating system is Windows 10. Apache/PHP is running inside VirtualBox under Ubuntu 16.04 LTS. The Ubuntu virtual machine and host Windows system are connected to the same virtual network, so the host OS can access Apache in virtual machine. Host OS can access Apache on port 8085.
Some supporting applications for text parsing are going to be used during testing to extract sequence data for the toolkit. These are running under Linux subsystem for Windows.
The whole software stack is running on my laptop with Intel i7-2640M CPU with 2 cores and 4 threads at 3 GHz. There is 16 GB of available RAM and 500 GB SSD is used for storage.

4.2 Basic Functionality tests

In this section, a simple PHP script was generated, that generates and outputs two UUID. This section will be used to validate functionality of almost all toolkit components.
The UUID is created using the code mentioned in the Design chapter. The whole script looks like this :

```
<?php
function UUIDv4(){...}
echo UUIDv4();
```

```
?>
```

4.3 Apache Process Pwner Test

First, the target sequence must be obtained from the server. If the attacker would simple download the UUID from the server, he would probably not get output from freshly seeded PRNG. In order to get data from freshly seeded PRNG, the `overload` command of the toolkit is executed:

```
>>overload 127.0.0.1:8085/toolkitTest1.php -n1 15 -n2 15
    -o uuidOut
```

Before executing this command, the following bash command can be used on the target server to check currently running Apache processes:

```
$ps aux | grep apache
```

Running this command reveals 10 running processes on my configuration of Apache. The process IDs are 4944-4953. Now the overload command can be executed and while it operates, the Apache process manager behavior can be observer by running the ps command repeatedly.
During the execution of the command, a sudden spike in number of running Apache processes was observed, as expected. After the command finished, the ps command was executed for the final time. It showed that again only 10 Apache processes were running – this time with process IDs 5045-5054, confirming that Apache has indeed created 10 new processes and killed the old ones.
On the SSD were added 15 new files, named uuidOut1.txt to uuid15.txt and each contains some UUID, which allows the simulated attack to progress.
This confirmed the functionality of the APP component. If the APP would not be able to create new processes, number of connections can be adjusted to get the right results.

4.4 PRNG Cracker Test

During this test, functionality of the cracker is tested and it is going to advance the attack started in previous section.
First, input data is prepared. The toolkit expects the input sequence in the command line input or as a single file. In this case, input from the file is tested. To create the input file, a very simple bash script is executed which just joins the 15 files from previous section into one file.
After this, the cracker must be told, what kind of data is it going to crack. This is done using this command:

```
>>format uuid1
```

The format parser component of the toolkit responds to this message as expected :

```
Accepted new formatting : [UUID1]
```

Now, the cracker is told to load file named `uuidOutputs.txt` and the cracker tries to find seed to each line of the input.

```
>>loadcrack uuidOutputs.txt
```

After running this command it can be seen that the seed was successfully found for each PRNG output, confirming the ability of the toolkit to crack the sequences.
The toolkit also reports the time it took to crack each sequence. It was 2000 ms for each input, so the cracking took 30 seconds it total.
In this case, the cracker not only found the seed for each PRNG that is currently running on the Apache server, it has also replicated the PRNG with the same state in the memory, which will be showcased in next section

4.5 PRNG Explorer and finder

Let's now say some victim user calls this UUID-generating script and receives an UUID, which the attacker would like to know. Even though the attacker knows the internal state of each PRNG on the server, he doesn't know which PRNG the user accessed. Luckily, the toolkit can reveal this information.
First, it gets output from each Apache process again, this time without spawning new processes. This is done by setting first parameter of `overload` to 0, so output from each request is saved.

```
overload 127.0.0.1:8085/toolkitTest1.php -n1 0 -n2 15 -o
    uuid2Out
```

Next, this outputs are again joined together into single file. The attacker is then going to test new command : `loadfind`. The command loads the file, and searches each in-memory PRNG if it generates given output. The output looks something like this :

```
...
Output found ! Seed = 1980426548, position = 8, generator
    ID = 2
Output found ! Seed = 1411765300, position = 8, generator
    ID = 3
Output found ! Seed = 2906512837, position = 16,
    generator ID = 5
Output not found...
Output not found...
...
```

In the excerpt from the console, it can be seen that for two outputs, the output couldn't be generated by any internal PRNG. This is because the `overload` was accessing more processes then Apache was running, so Apache had to create new processes which were not cracked yet.
First two outputs were found in generators with IDs 2 and 3. It can be seen that generators created 8 outputs since seeding. This is correct, since one UUID needs 8 mt_rand() calls.
However, the generator number 5 has been used for 16 outputs. And since the attacker was not using the server in the meanwhile, this is the generator the victim used for creating his secret UUID. So the only thing left, is to generate two UUIDs – first will be the one the server generated in the first call of `overload` command and the second will be the secret UUID token. Generating output from each in-memory PRNG is easy and it's done with the `generate` command. After testing the whole process locally I was able to check that the toolkit indeed found the secret UUID !
This last section confirmed the function of PRNG Explorer and generator, since it successfully found and generated the same UUID as PHP server. By performing this whole attack, each component of the toolkit was tested and shown to be functional. PRNG vulnerability exploitation using this toolkit was also showed feasible.

4.6 Speed comparison with other software

In this section, I will compare cracking speed with other software solutions. For this, I have selected php_mt_seed and Untwister from the Analysis chapter. I will show few scenarios, where my solver is faster, and scenarios where my solver is getting slower.

4.6.1 Cracking native mt_rand() output

In this test, the PRNG is manually seeded with mt_srand, so the seed is known for sure. The job for the crackers is to crack the seed of the following output :

```
<?php
mt_srand(123456789);
echo mt_seed();
?>
```

The output of the script is 997730858. This result actually has 3 different seeds and the solver is expected to find them all. The results are following (the toolkit in this thesis is named rm_toolkit) :

- php_mt_seed : Running at 100 %, fully utilizing all CPU cores, the solver ran for 2 minutes and 40 seconds. It fully recovered all 3 seeds.

- Untwister : Never finished. Expected time to finish the computation was 32 hours, so the computation was terminated manually.
- rm_toolkit : Less than 1 ms. The computation finished almost instantly and successfully found all three seeds.

4.6.2 Cracking bounded mt_rand() output

In this section, mt_rand() output is bounded, so rm_toolkit has to make more lookups in the rainbow table. The script used was following :

```
<?php
mt_srand(123456789);
echo mt_seed(0,1000000)." ".mt_seed(0,1000000);
?>
```

This script generated two numbers : 464605 and 11913. All three solvers were supplied with information about the target range of 0 to 1000000. The results are following :

- php_mt_seed : The solver ran for the same time of 2 minutes and 40 seconds. The computation ran for equal time, since it was required to make the same number cycles, it simply checks all the seeds. The seed was recovered successfully.
- Untwister : Again, the solver failed to finish under one day.
- rm_toolkit : the computation ran for 120 ms. This is much slower than in first case, but still much faster than other solutions. The seed was recovered correctly. In order to tell the solver what to solve, these two commands were used :

  ```
  >>format n,0-1000000 n,0-1000000
  >>crack 464605 11913
  ```

If the range of the mt_rand() was bounded even more, the php_mt_seed would eventually become faster. However, the whole rainbow table could be moved from SSD into RAM, making it again a few orders of magnitude faster. Since I currently don't have access to a machine with enough RAM, these tests were not completed).
These tests showed, that in many cases, rm_toolkit is much faster than other solutions.

Conclusion

The goal of this thesis was to implement a set of tools for easier and faster exploitation of PRNG vulnerabilities.
Maximum effort was given into fulfilling these goals, but only time and user reports can prove this. However, the toolkit successfully implements several techniques discussed in the Analysis chapter. The toolkit also implements all the functional requirements, satisfy all non-functional requirements and all the use cases presented in the Design are executable and working as previously tested.
The toolkit implements a new technique for PRNG cracking using rainbow tables. This technique is much faster in many scenarios as shown in the Testing chapter. By loading the rainbow tables into memory, even bigger speed-up can be achieved. Other improvements could be later made – per application table creator, more advanced cracking alorithms and so on.
There is a lot that can be improved in this toolkit,since this is such a huge topic. Automatic input parsing is one of the ideas – the toolkit would include advanced heurictic and it would be able to parse the input without the previous `format` commands.
The toolkit also focuses only on PHP's PRNG. Crackers for other platforms, such as Java, .NET, glibc, JS and others could be easily added and it would further aid pentesters in their work.
I hope somebody finds some uses for techniques described in this thesis, since they have the potential to finaly aid in birth of fully universal PRNG cracking toolkit. Personally, I have found this topic to be very interesting and I hope more attention would be given to it.

Bibliography

[1] Linear congruential generator. 2017. Available from: `https://en.wikipedia.org/wiki/Linear_congruential_generator`

[2] Definition of randomness in English:. 2017. Available from: `https://en.oxforddictionaries.com/definition/randomness`

[3] Davis, P. J. *Mathematics and common sense.* Wellesley, Mass.: A.K. Peters, 2006th edition, 2006, ISBN 15-688-1270-1.

[4] Random Class. 2017. Available from: `https://msdn.microsoft.com/en-us/library/system.random(v=vs.110).aspx`

[5] Quantum Random Number Generators. 2017. Available from: `https://arxiv.org/abs/1604.03304`

[6] The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. 1995. Available from: `http://stat.fsu.edu/pub/diehard/`

[7] A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. 2010. Available from: `http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf`

[8] What & how is MT? 2016. Available from: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ewhat-is-mt.html`

[9] Problems with Linear Congruential Generators. 2016. Available from: `https://cs.adelaide.edu.au/~paulc/teaching/montecarlo/node106.html`

[10] *Machine Learning: An Algorithmic Perspective.* Chapman and Hall/CRC, first edition, 2011, ISBN 9781439889213.

[11] Random (Java Platform SE 8). 2017. Available from: https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

[12] Knuth, D. E. *The art of computer programming*. Reading, Mass.: Addison-Wesley, third edition, c1997, ISBN 02-018-9684-2.

[13] Reference Source. Available from: https://referencesource.microsoft.com/#mscorlib/system/random.cs

[14] Lcg_value. 2001-2017. Available from: http://php.net/manual/en/function.lcg-value.php

[15] PHP 7 ChangeLog. 2001-2017. Available from: http://php.net/ChangeLog-7.php#7.0.0

[16] Openssl_random_pseudo_bytes. 2001-2017. Available from: http://php.net/manual/en/function.openssl-random-pseudo-bytes.php

[17] Comparison of the usage of Apache vs. Nginx vs. Microsoft-IIS for websites. 2009-2017. Available from: https://w3techs.com/technologies/comparison/ws-apache,ws-microsoftiis,ws-nginx

[18] Argyros, G.; Kiayias, A. I Forgot Your Password: Randomness Attacks Against PHP Applications. 2012. Available from: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/argyros

[19] Langkemper, S. Cracking PHP rand(). Available from: http://www.sjoerdlangkemper.nl/2016/02/11/cracking-php-rand/

[20] Php_mt_seed - PHP mt_rand() seed cracker. Available from: http://www.openwall.com/php_mt_seed/

[21] GitHub - altf4/untwister: Seed recovery tool for PRNGs. Available from: https://github.com/altf4/untwister

[22] Using Web Service Credentials. 2017. Available from: https://marketing.adobe.com/developer/ja/documentation/authentication-1/using-web-service-credentials-2

[23] Amazon EC2 Pricing. 2017. Available from: https://aws.amazon.com/ec2/pricing/on-demand/

[24] SecTheory. 2017. Available from: https://samsclass.info/seminars/slowloris.pdf

Appendix **A**

Acronyms

GUI Graphical user interface

CLI Command line interface

RNG Random number generator

APP Apache Process Pwner

PRNG Pseudo-random number generator

PHP PHP: Hypertext Preprocessor

MT Mersenne Twister

CPU Central processing unit

RAM Random access memory

SSD Solid-state drive

HDD Hard-disk drive

UUID - Universally unique identifier

AWS - Amazon web services

AVX2 - Advanced Vector Extensions 2

LCG - Linear congruential generator

NIST National Institute of Standards and Technology

Appendix B

Contents of enclosed memory card

- readme.txt ... the file with memory card contents and project description
- rm_toolit_solution ... the directory with project solution
  - rm_toolkit_APP ... Apache Process Pwner project directory
  - rm_toolkit ... main toolkit project directory
    - bin ... compiled binaries
- thesis_source ... the directory of LaTeX source codes of the thesis
- text ... the thesis text directory
  - thesis.pdf ... the thesis text in PDF format
- tests ... Directory with some of the scripts used for testing