JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

App Store for Massive Scale Software Diversity

Dominik Lichtenauer

Preliminary Version of a Master's Thesis
to be submitted to the
Johannes Kepler University Linz, Austria

produced in cooperation with the
Secure Systems and Software Laboratory
University of California at Irvine

funded by the Austrian Marshall Plan Foundation

Advisors:
Prof. Dr. Michael Franz (UC Irvine)
Prof. Dr. Hanspeter Mössenböck (JKU Linz)

Linz, November 22, 2012

Abstract

The development of modern software became a complex business which makes it almost impossible to eliminate all bugs. Attackers take advantages of these existing errors to exploit their target. The current situation of widespread distribution of the same binary facilitates their malicious intention because their attack is likely to succeed on many computers. A project launched by the University of California in Irvine has the aim to break this software monoculture with massive scale binary code diversity. Beside the diversification engine, the creation and distribution of the unique software packages is a key component. This paper describes an approach by using Amazon's cloud services where the diversification process is entirely transparent and makes no change to the developer nor to the consumer.

1

Introduction

1.1 Motivation

Computers and network devices are under constant attack from a variety of adversaries. They use vulnerabilities such as errors in shared libraries, application programs and operating systems to perform unauthorized operations. Today's modern software is too complex to consequently eliminate all bugs, even if a lot of advances have been achieved in the past. The existing of these errors becomes a threat when simultaneously millions, or in some cases hundreds of millions of computers are affected by the identical vulnerability at the same time - which is the situation today. The situation of widespread distribution of the same binary ("software monoculture" [GPS+03]) makes it attractive for an attacker, because the build of one exploit is likely to succeeds on a large number of targets. Additional, the adversary can replicate the target environment to use it for debugging and off-line testing while developing the attack.

Despite desktop systems, the increase of power and the rapid extension of modern smartphones represent another potential target for attackers. The operating systems of mobile devices are frequently based directly on desktop operating systems (such as Linux) and their running applications on top of them are starting to grow in size. It will be just a matter of time until smartphones face the same exploitable vulnerabilities as their larger desktop counterparts.

An approach to increase the cost of attackers is the use of massive-scale binary code diversity. For this reason, the University of California in Irvine launched a project to automate this diversification process without a change, neither to the developer nor to the end user [JHC+12]. The idea of massive scale software diversity is, that on every computer runs a functional identical, but unique binary that is automatically generated by compiler-based techniques ("multicompiler").

Instead of producing a single "optimized" binary, the diversification engine generates many alternative binary executables from a single source program. This technique will not remove the vulnerability of the program because of programming errors, but it makes it much more difficult to exploit them, especially an attack procedure called "Returned Oriented Programming". As a result of this, a single attack payload will only succeed on a small fraction of the potential target systems. To compromise a large number of targets, many different attacks will be necessary without a priori knowing which attack variant needs to be directed to which particular system to succeed.

This approach of code diversity does not only remove the problem of broad scale attacks, in which a single exploit package (such as worms and viruses) ripples through a large number of network-connected systems in a very short time, it also migrates directed attacks by adversaries, even those who have access to nation-state scale resources. This is because there is a very low probability of correctly guessing the particular version of the diversified binary is installed on the target system. Moreover, this technique makes targeted attacks economically unviable. Absent an insider with the knowledge of the specific binary running on the target communicating with the attacker, the attacker would not know which large number of payloads are going to succeed. The only way is brute-force this problem by using a large number of different attack vectors. Therefore substantial time would be required to create them and the resulting amount of network traffic increases the likelihood of detection.

Another important aspect is that the way of creating attack vectors by reverse engineering of security patches will become much more difficult. To make a successful exploit with this technique, two pieces of information are required to extract the vulnerability from a bug fix: the version of the application and the specific patch that fixes it. In an environment that uses unique instances of each application, things can be set up so that the attacker never obtains the vulnerable application and the corresponding patch to identify the vulnerability.

1.1.1 Paradigm Shifts

The multicompiler project is enabled by four fundamental paradigm shifts that have happened in the past years [TJ]:

1. **The Way of Software Delivery:**
   Not so long ago, software was delivered on CDs or DVDs which makes it impractical to give every user a unique version. The change to download

software makes it possible to provide a different version with the exact same functionality to each user. From the perspective of the user nothing changed but for the attacker things become harder.

2. **Ultra-Reliable Compilers:**
   Until quite recently, compilers were the most complex software program and it wasn't unreasonable to assume that they might contain errors itself. One consequence of this was the focus on software certification and the testing of the end-product. This made the compilation to a very predictable and reliable process, bugs in software that can be traced back to compiler errors are very rare. Just-in-time compilers are now widespread and the output is executed by millions of users without any further testing. Another fact is that compilers today are not more complex than they were 20 years ago, unlike almost all other software. Their code has been refined over decades rather than enlarged.
3. **Cloud Computing:**
   Generating a unique software version for each user would have been impossible in the past because of the required infrastructure - the costs would have been exorbitant. The use of cloud computing platforms makes it possible to scale up resources instantaneously to react to changing demands. Furthermore, there is need for an up-front investment and the costs per unique binary is constant, no matter how much versions are created.
4. **"Good Enough" performance:**
   Modern computers provide enough performance in most domains and users are willing to accept small performance losses for additional security.

1.2 Definition of objective

One of the challenges of the multicompiler project is the distribution of the different versions of a software program to the end-user. The whole diversification process has to be transparent to the consumer and poses no extra effort to the software developer. The goal of this master thesis the design and implementation of an "App-Store" that meets this requirements. Major problems to solve are to generate enough diversified binaries, prevent the user from waiting for a download and to make the delivery process secure.

Producing a large amount of unique binaries requires a lot of infrastructure. To keep the costs low, it is rational to use cloud technology. So an important part for this work was to find a suitable cloud service. In the recent past years, cloud

computing has seen explosive growth and is now available from many commercial providers. Among the choices were the well-known IT companies Amazon, Google, IBM and Microsoft.

Important aspects for the decision were:

- Performance
- Easy to use
- Available operating systems
- Pricing
- Services
- Support
- Reliability

In order to meet these requirements, Amazons cloud service was selected. The reasons were:

1. Amazon has a long experience in offering large-scale, global infrastructure which makes them reliable.
2. The pricing is moderate and no long term commitment is required.
3. Besides compute power and storage, a lot of other services (such as databases and messaging) are available.
4. Several different compute instanced can be used.
5. All services are accessible with different programming languages.
6. The start with Amazon's cloud environment is made easy because many demos and tutorials are offered.

As an example to demonstrate that the concept of building a large amount of unique software is possible, diversified versions of *Googles Chrome* are build in the cloud and as download on a website available.

2

Return oriented programming

2.1 Introduction

In order to create a successful exploit, two steps are required: First, the attacker has to find an entry point to inject a malicious payload. In the second step the control flow on the target system has to be redirected to the injected payload. In the beginning of software attacks the injected payload was executable code. Invented defense mechanism such as Microsoft's Data Execution Prevention or non-executable memory stopped this types of attacks. In 2007, Shacham presented technique known as *Return Oriented Programming (ROP)* to bypass these existing defense mechanisms [Sha07]. Instead of injecting new code, this technique uses already existing code snippets in the target and is in practice just as powerful as direct code injection attacks. Before the explanation of how a ROP attack works, an overview of the evolution of return oriented programming is provided.

2.2 History of ROP

2.2.1 Buffer overflows

Buffer overflows exist for more than 30 years and occur in the original form because an error due insufficient bounds checking. This means that more data is written into a buffer than it can store, as a consequence it corrupts data values in memory addresses. Most of this errors occur when strings or characters are copied from one buffer to another. An example is shown in Listing 2.1, if the command line argument is larger than nine characters the program will result in corruption.

A common technique to raise a buffer overflow is to attack a buffer on the stack. A simple way is to inject a string which is actually an executable binary code. Another is to change the return address to a point where malicious code is stored in memory. To make a successful attack, some challenges like null bytes in addresses need to be overcome to make them reliably.

```
void vulnerable_func(char* c)
{
    char buffer[10];
    strcpy(buffer, c);
}

int main (int argc, char **argv)
{
   vulnerable_func(argv[1]);
}
```

Listing 2.1: Example of a vulnerable C program

2.2.2 Return-to-libc

For this attack, usually a buffer overflow vulnerability is used to hijack the control flow. But instead of injecting new malicious code, the attacker calls an already existing function from a library. By executing one malicious instruction in system mode it is possible to make arbitrary computations from existing code. When the attacker gained control, the return address on the stack is replaced by another address that points to a useful function. Additional, the attacker has to set up data in the memory in a way to provide arguments to the function. This technique is called return-to-libc because libc is a popular target (it provides useful functions like *system()* or *exec()*) even if completely different libraries are used.

2.2.3 Borrowed code chunks technique

With the introduction of application binary interface (ABI) on x86-64 Linux systems, the return-to-libc did not work anymore because function call parameters have to passed within registers instead of putting them on the stack. In 2005, Sebastian Krahmer [Kra05] introduced a new approach called “borrowed code chunks technique” to bypass this protection mechanism. He uses already existing code sequences in the target to move the parameters from the stack into the register. When the correct values are present in the registers, it is possible to call the desired function like in the return-to-lib technique. Furthermore, Krahmer developed an automated exploit generator that searches and resolves the required code sequences and symbols in the address space to make the attack easier.

2.3 Return oriented programming

The generalization of the return-to-libc exploit is return oriented programming. Return oriented programming allows an attacker to perform arbitrary code execution without injecting new code into the target program. Important for ROP is the concept of *gadgets* which are short byte sequences that end with a return (or functional equivalent) instruction. After the attacker has control over the stack (for example through exploiting a buffer overflow), the starting address of the gadgets are placed on it to be executed in a sequence. An example attack is shown in Figure 2.1.

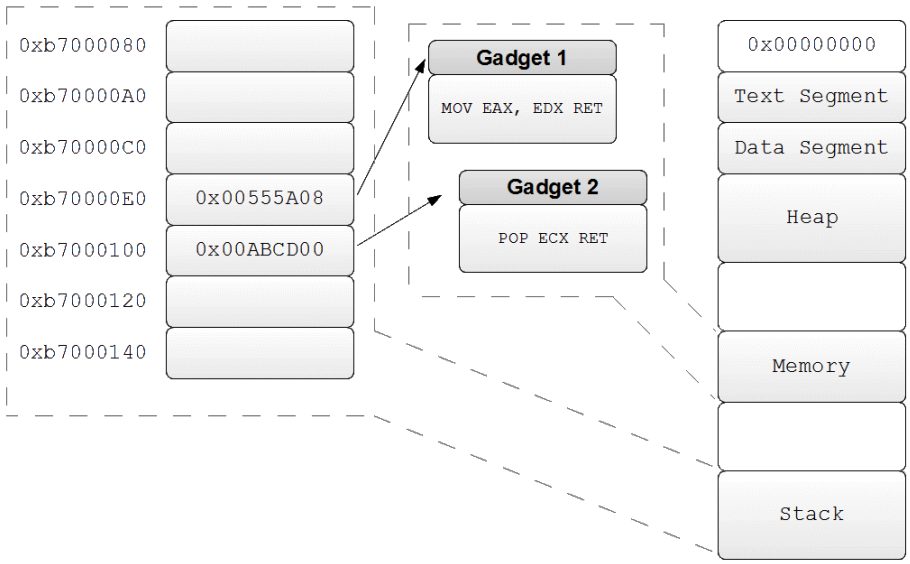

Figure 2.1: Example of return-oriented programming

A return instruction has the following consequence:

1. the instruction pointer retrieves the value from the top of the stack, this has the effect that the instruction beginning at that address is executed
2. the value of the stack pointer is increased

This enables the use of the stack in an unconventional manner, because the return instruction at the end of a gadget causes a return to the next one. The result is, that each gadget is executed one after another.

Essential for ROP is to analyze the target environment (libraries and programs) to build a gadget catalogue. On any large binary distribution, enough useful instructions can be found to build a Turing-complete set of gadgets.

2.3.1 Automated return oriented programming

The next step is to automatize ROP by using a “gadget-compiler”. The idea is that the compiler automatically finds gadgets on the target system and combines them in a way that they are equivalent to a given source program (for example in C).

3

Multicompiler

3.1 Introduction

The idea of software diversity is not new, but in practice it has only recently become feasible to create a unique binary for each user. Essential aspects are the reproducibility of the unique binaries and that the diversification process is entirely transparent to the developer and to end user.

The approach of the multicompiler project is a source-code oriented diversification engine inside the the *Low Level Virtual Machine (LLVM)* framework by adapting its back-end with diversification techniques. LLVM is collection of compiler technologies containing tools (for debugging, optimizing), front ends and a test suite. As a consequence of using this system, all supported input langues of LLVM can be used [JHC+12].

Traditional compilers use heuristics and algorithms that are deterministic which has the effect that they will always make the same choice providing the same input. The multicompiler instead enumerates possible choices in order to produce outputs that are different from each other.

Compile-time diversification enables to diversify any existing software on any level of a system. Depending on the desired security, operating systems, system software and application can be adapted.

This chapter shows available defense methods and explains in more detail those, which are currently implemented in the multicompiler. Furthermore the effect of the security - performance tradeoff and the seed generator are discussed. Other interesting aspects of the diversification process are the impact on the executable size and on the build time. But test have shown, that both considerations have negligible effect on the result [Jac11].

3.2 Available defense mechanisms

This section discusses available diversification variants to modify an executable [TJ]. This can happen at compile time as well as by manipulating the binary code. Some of this techniques are ineffective when used alone, but become powerful when they are combined. A more detail explanation of NOP insertion, equivalent instruction substitution and instruction scheduling is presented in Section 3.3

- **Stack cookies:**
  Stack cookies is a prevention technique that adds some code generated by the compiler to a function to detect buffer overruns which overwrite the return address. When an application starts, the cookie (special random value) is created and copied to the stack upon a function entry. During the function epilogue, the code checks if the value is unchanged. If the value is different, corruption of stack frames are detected and the program will terminate. Stack guard is an example that uses this technique to detect stack smashing attacks.
- **Stack base randomization:**
  This protection method is implemented in most modern operating systems. When an application starts, the stack starts at a different base address.
- **Stack frame padding:**
  Stack frame padding extends the length of the stack frame to prevent stack based buffer overflows. With this method, a stack based buffer overflow can not successfully exploit the target because the injected payload is not large enough to overwrite the return address. One implementation techniques is to add dummy stack object.
- **Reverse stack:**
  On common processers architectures the stack grows in one direction, for example, the Intel x86 instruction set has a downward growing stack. A possible way to make the stack grow upward is by augmenting stack manipulation operations with substraction and additions.
- **Heap layout randomization:**
  Corresponding to the stack based randomization, the heap layout can be randomized to make heap overflow attacks inefficient.
- **Variable recording:**
  This approach increases the effectiveness of stack cookies. When stack cookies are enabled, the attacker is still able to overwrite local variables which are located between a buffer and the cookie on the stack. To avoid

this, buffers are placed right after the cookie and other variables. Also copies of parameters of a function are placed after all buffers.

- **Equivalent Instructions:**
  Most hardware architectures offer several instructions that have in particular cases the same effect. Replacing such instructions with equivalent ones have no loss in performance but the binary sequence is changed significantly.
- **Instruction scheduling:**
  Usually, instruction scheduling is used to improve performance. Rearranging the order of instructions has the welcome side effect, that start addresses of instructions change and as a consequence gadgets eliminate.
- **System call number randomization:**
  Attacks that use directly encoded system calls must know the according system call numbers. When these numbers are changed the exploit will execute a different system call and this leads to unknown behavior. But there are two disadvantages: system call numbers are limited and brute force attacks are possible, also the kernel has to know the new numbers.
- **Register randomization:**
  Most exploits rely on fixed values in registers. A simple way to prevent these attacks is to exchange the meaning of two registers, for instance, the stack pointer register on x86 architectures *esp* can be exchanged with a general register like *ebx*. If the attacker places a system call number in *ebx* the execution will fail because the system will take the value from *esp*. The problem is, there is no hardware architecture that provides this mechanism. As a result it is necessary to exchange the registers before instructions that rely on values of these registers are executed.
- **Library entry point randomization:**
  This approach randomizes library entry points as a defense mechanism to prevent calls into libraries. There are two possible ways to perform this: rewriting function names during load time or rewriting them in the binary. The second one has the advantage that it has to be done only once.
- **Code sequence randomization:**
  To change generated machine code, call inlining, loop distribution, instruction scheduling and many other compiler transformation techniques can be used. The effect of this randomized code is that ROP become much more difficult.

- **NOP insertion:**
  NOP insertion is another technique to make ROP exploits harder. It is similar to stack layout randomization and stack frame padding. But instead of doing the operations on the data level, they are done on the binary code level. *No-operation (NOP)* instructions are short code fragments without a practical effect when executed. They can be used as a padding in the code to push the next instruction forward by a few bytes.

- **Code and data separation:**
  The aim of code and data separation to is prevent the execution of injected code. The idea is, that certain memory areas can never be executable and writable at the same time. AMDs solution is the NX flag (No Execute Bit) to avoid that non-privileged code can access privileged code and data. The corresponding technology of Intel is the XD Bit (Execute Disable Bit). If supported by the operating system, the NX/XD bit marks a memory page as data only or as executable. This hardware mechanisms help to prevent malicious attacks when combined with operating system that implement this technology. For example, on Microsoft Windows this feature is called Data Execution Prevention (DEP).

- **Address space layout randomization:**
  Advanced space layout randomization (ASLR) is another security technology that makes it more difficult to exploit vulnerabilities. Many techniques try to overwrite return addresses with static pointers to malicious code. ASLR moves addresses of executables, libraries, stacks and heaps in memory to random locations to make it difficult to predict the target address. Most modern operating systems such as Windows, Moc OS X and Linux have implemented this approach.

- **Program section and function recording:**
  Modern programs are created by putting together several modules, these modules are divided into sections that contain functions. Some types of attacks rely on the knowledge of the location of a certain function, so an option to avoid these behavior is to reorder the functions themselves at a local level and the code during the linking stage.

3.3 Implemented diversification techniques

This section describes effective methods for compiler based code diversity. The focus is on methods that do not interfere with program execution.

3.3.1 NOP insertion

Gadgets are usually found in the middle of another instruction. The x86 instruction set has a variable length and the encoding depends on the first byte of the instruction. Inserting a *NOP* instruction within a gadget has no effect during execution but it can eliminate or significantly alter the gadget. Figure 3.1 demonstrates this effect [Jac11].

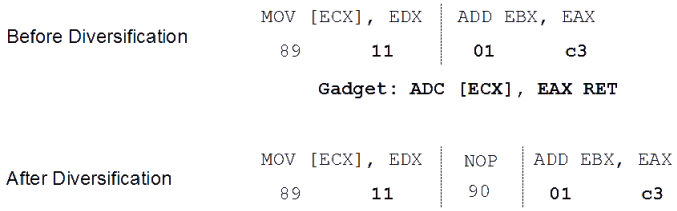

Figure 3.1: NOP insertion example

The NOP insertion transformation is implemented as LLVM Pass in the x86 backend. In order to decide how often a NOP is inserted, a probability value *pNOP* from 0 to 100 is passed to the compiler as command line argument. During the compilation process, for each instruction a random integer between 0 and 100 is generated. If this value is less than *pNOP* x 100, a NOP is inserted in front of the current instruction, otherwise the pass continues with the next instruction and repeats the trial.

3.3.2 Instruction scheduling

To increase performance in in pipelined architectures compilers rearrange the order of instructions. The diversification engine manipulates the scheduler's input to change the location of gadgets to neutralize them. Figure 3.2, shows an option by reordering the *MOV* and the *ADD* operation [Jac11].

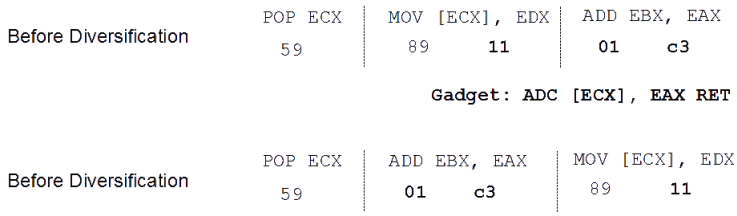

Figure 3.2: Eliminating a gadget by instruction scheduling

For performance reasons, LLVM uses architecture specific scheduling algorithms, on x86 targets, a priority queue scheduler is used. The modifications for the multicompiler are applied to the pre-register allocation instruction scheduler. Instead of using the instruction with the highest priority, a random one is selected.

3.3.3 Equivalent instruction substitution

Most hardware architectures provide several instructions with the same functionality. This enables to replace one such instruction with another equivalent instruction with non-essential impact on performance, but changing the underlying bytes that compose gadgets. In the multicompiler, *MOV* instructions are replaced with the equivalent *LEA* instructions. These operations are used to move values between registers. The implementation is similar to the NOP insertion design, the probability *pEquiv* is passed as command line parameter. A *MOV* instruction is replaced with a *LEA* instructions when the generated random number is less than *pEquiv* x 100.

3.4 Performance and security

This section reports a short summery on the impact of performance and security of the diversifying compiler. Details about the benchmark test suite, the system configuration and the test procedure are presented in [Jac11]. For the evaluation, all implemented techniques have been tested sperate and in combination. Instruction scheduling was always enabled *ISched(1)*, an additional option was *ISched(2)* with the purpose to use the worst possible instruction arrangement. NOP insertion *Nop* and instruction substitution *ISub* have been tested with the values 50 and 100.

Operations	Performance overhead in %	Remaining gadgets in %
ISched(2)&Nop(100)	19	0,295
ISub(100)&Nop(100)	19,5	0,305
ISub(50)&Nop(100)	19	0,31
Nop(100)	19	0,315
ISched(1)&Nop(100)	20	0,33
ISub(100)&Nop(50)	6,5	0,35
ISched(1)&Nop(50)	8	0,37
ISched(2)&Nop(50)	7,5	0,385
Nop(50)	7	0,4
ISub(50)&Nop(50)	7	0,4
ISched(2)&ISub(100)	1,5	0,9
ISched(1)&ISub(100)	1	1,05
ISched(1)&Isub(50)	0	1,15
ISched(2)&ISub(50)	1	1,23
ISched(1)	0	1,5
ISched(2)	1	1,72
ISub(100)	0,1	3
ISub(50)	-0,15	6,5

Table 3.1: Summary of evaluation

Table 3.1 shows that there is a tradeoff between security and performance. But even with a very low performance overhead around one percent, only a small amount of gadgets remain in the binary file. NOP insertion generally dominates the performance impact, especially when combined with a second transformation. The best security related results are obtained with worst case instruction scheduling in combination with NOP insertion.

3.5 Random seed generator

The random seed is a key component of the diversification engine. The seed ensures that the paths chosen by the multicompiler during the diversification process are reproducible. A random seed generator creates a state file that contains a random number sequence which is passed as a command line argument to the multicompiler. This file is a shared secret between the App Store and the client to enable update and debugging mechanisms.

The generator has to meet the following requirements for the diversification engine:

- Inside a software package, the seed has to be valid for all compilation units
- Without the seed, the identical binary cannot be recreated

- Re-seeding has to happen in a secure way
- At no moment, two random seeds used inside the compilation process should overlap
- A seed must always produce the same result
- The computational overhead should be very low
- The client can use the software without any additional steps

The current design of the generator is cryptographically secure and is based on the design specified in NIST SP 800-90 [1]. The state file contains in the first stage the initial state of the generator, in all other cases the last state. The multicompiler initializes the pseudo random number generator (PRNG) based on the content of the seed file. When the compilation is done, the state of the PRNG is updated to the file. In the current design, AES with a 128-bit key size is used as a pseudorandom function, but is extensible to other block ciphers.

[1] "This Recommendation specifies mechanisms for the generation of random bits using deterministic methods." [BK12]

4

Design of the App-Store

4.1 Introduction

This chapter describes the Design and Architecture for the App-Store for the distribution of the diversified binaries. The whole diversification process is entirely hidden in the cloud and transparent to the consumer and to the developer (shown in Figure 4.1). The end user visits a website - the App-Store - and downloads the desired application. The developer submits the source code, the instructions for the build process and the associated dependencies such as required libraries and software packages to the administrator of the App-Store.

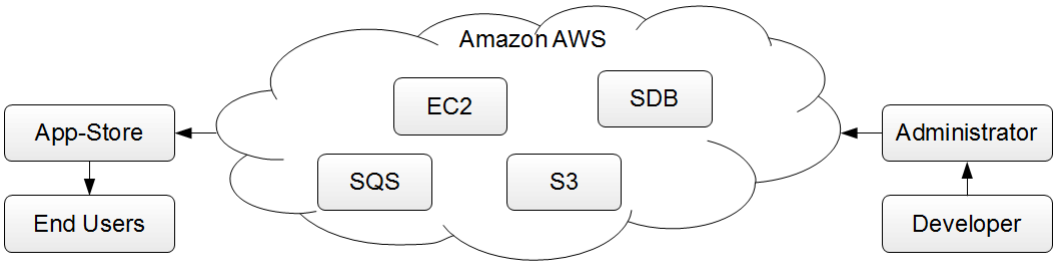

Figure 4.1: Basic overview

There are two reasonable solutions for building a large amount of unique software versions:

- The fist approach is the more simple one: A certain amount of unique instances of an application is created and stored permanently on Amazon S3. When an end user request a download, a random generator chooses one of the diversified binaries and offers it to download. The drawback of this

solution is to predict the number of how many different versions should be created for a specific application. There are many software packages which are downloaded millions of times, so if for example only 1000 unique binaries are created, many users will receive the same piece and the probability that an exploit will work on many targets is increased a lot.

- The focus of this project is on security which leads to the second and chosen possibility: for each user a unique version is build. The disadvantages of this way is that a single download becomes a little bit more expensive but still stays very moderate. The main problem to handle is to have all the time enough downloads available.

To guarantee the requirements, four of Amazon's cloud services have been chosen and connected in an effective manner. Amazon's Simple Database is used to store all configurations, keys and build settings. With Elastic Compute Cloud, the unique binaries and the random seeds are generated. In the Simple Queue Service, download links and seed locations are saves temporary. And on the Simple Storage Service, the diversified applications and state files are stored.

The design is divided into an administration and a client view, which are explained in the following sections.

4.2 Administration view

Figure 4.2 provides an overview, how the for the administration view required components are related and work together.

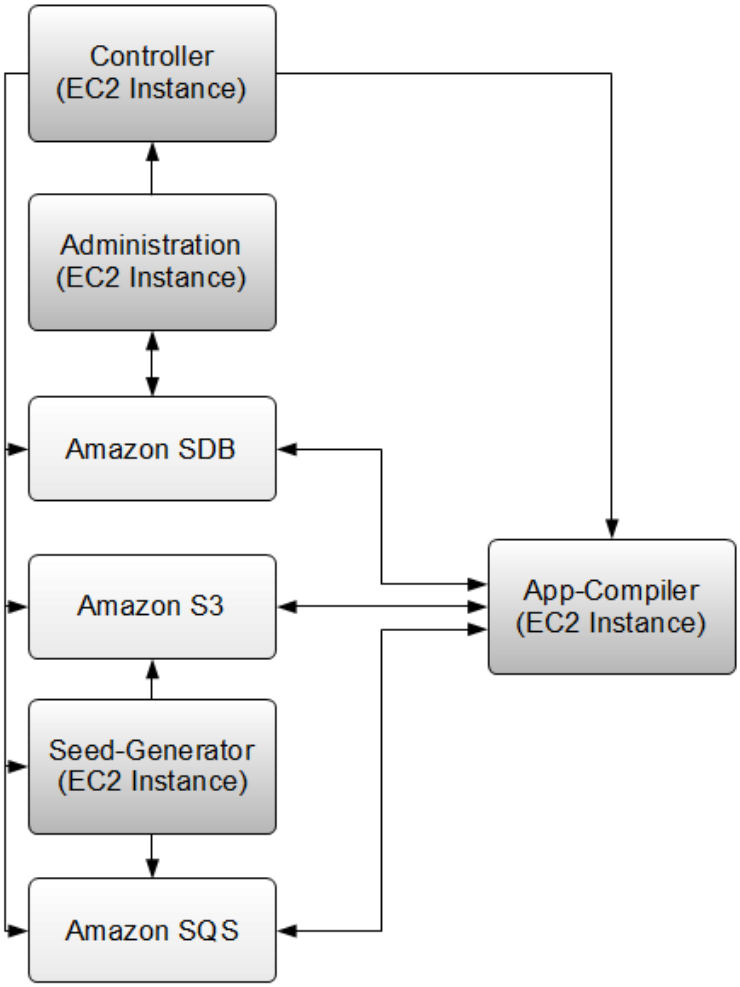

Figure 4.2: Administration view

4.2.1 Administration

The administrator is responsible for the initial setup of the provided software. There are some necessary things that have to be done before the creation of the unique application can be launched.

The first step is to create an Amazon Machine Image. The App-Store environment provides templates (dependent on the operating system) where the multicompiler

is preinstalled. With a simple script, the template is updated with the delivered software, saved and registered on Amazon EC2. Normally the creation happens automatically, but in some cases (for example large projects like Google Chrome), individual modifications need to be arranged.

The second step is to create the build task and provide some individual settings. This has to be done for each version that will be offered (see Figure 4.3). To distinguish between all tasks, a unique identifier is used which consist of the submitted build settings.

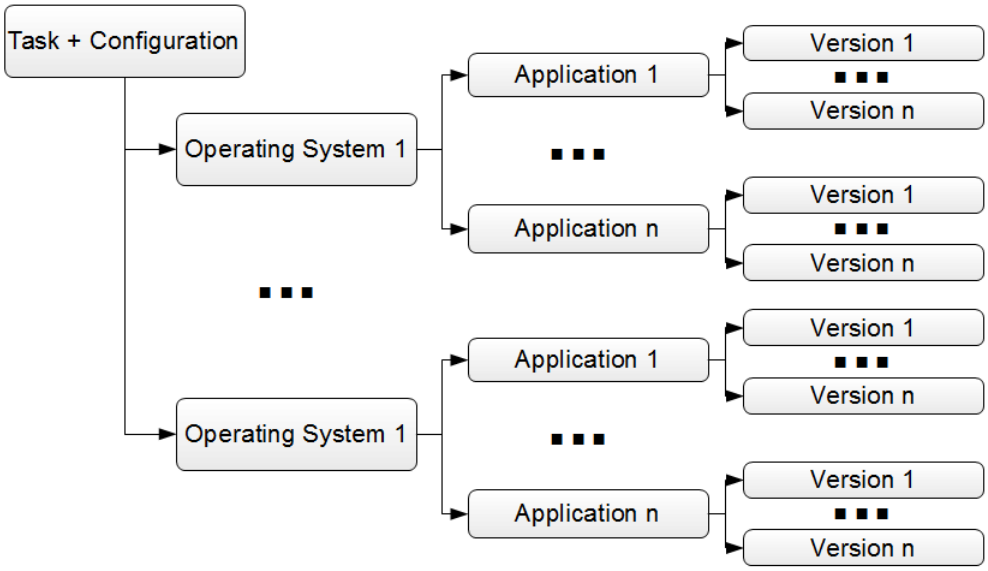

Figure 4.3:

The build task contains the information for the multicompiler and the installed main program on the AMI. The following parameters are required:

- **Parallel builds:** This option defines the number of parallel compilation instances per EC instance. The default setting is one.
- **Cores per build:** This parameter sets the number of how many cores are used for a compilation. This is only useful when the build process of the software is designed for it. If this is possible, the default value is the number of cores of the current EC instance.
- **NOP insertion:** Sets the probability for the NOP insertion (0-100).
- **Equivalent instruction substitution:** Defines the probability for the equivalent instruction substitution (0-100).

- **Instruction scheduling:** Determinants if this technique should be used or not.

The settings parallel builds and cores per build are strongly related and depend on the software to compile. To achieve the optimum, these parameters should be tested with different options.

Additional configurations settings are:

- **Name:** The name of the provided software.
- **Security Level:** Because the end user won't know anything about compiler transformations, the administrator has to set a security level that represents these settings. The options are low, medium and high.
- **Version:** The current software version.
- **Launch number:** Before the software can be offered as download, some versions have to be created in advance. This number defines, how many of them are created initially.
- **Minimum number:** This is a reserve of binaries that is permanently stored on S3. In the case, that the demand of a version increases dramatically within a short period, approach one, described in the beginning of this chapter, is used. But only until enough instances are launched to provide sufficient versions again.
- **Peak time:** It is normal that on a certain time of a day, more downloads are requested. This is for example during daylight (which of course depends on the region), but can be any other one for different reasons. The provided parameter is the time span in the 24 hour format.
- **Minimum number peak time:** This option is the same as minimum number, but for the peak time.
- **Update interval:** Defines, how often the controller adjusts the instances according to the download requests (in minutes).
- **Instance type:** Represents the instance type for this task (the parameters for this option are predefined)

It seems, that the administrator has a lot of work to do. But in practice, not more than two different versions of an application will be provided. One with a good security-performance tradeoff and maybe another one for high security demands.

Also, for most applications, default values can be used, the control program will make adjustments anyway depending on the demand.

When build and configuration settings are done, everything happens automatically, but if required, adjustment can be made everytime. If a version or an application is no longer needed, all the corresponding setup can be removed with one command.

4.2.2 Amazon SDB

Amazon Simple Database was chosen, because it is schema-less, so there is no need to define a structure of a record. Additional to the administration settings for each task, the following information is saved:

- **Current downloads:** To calculate, how many instances are required to produce the demanded versions, the current downloads (provided from the client) are stored. The update interval is specified by the administrator.
- **Running EC2 instances:** The public DNS of all running instances with a time stamp are logged. This is required to shutdown no longer needed instances.
- **Queue updates:** Sets the update time for queues (for the reason see 4.2.5.2)
- **Build time:** This options says how long it takes to compile the software (in minutes) and depends on the instance type. This value is created during the launch phase: The instances measure the compilation time and store it on a temporary table. At the end of this phase, the maximum number is taken as build time to be on the safe side.

For one task, three domains are created and use the following syntax: *operating_system.application.version.identifier*. To this sequence, settings, DNS and seed are appended. For example, the settings table for Google Chrome build on Ubuntu is then *ubuntu.chrome.23_0_1271_64.18501.settings*.

The third table contains the location for the secret seed and the build settings. Initially, the idea was to attach the seed to the diversified binary and the consumer receives both. But for security reasons, it is better to store the seeds on the App-Store. In this table, for each seed two keys are created which are universal unique identifiers. The first one is the location link on AmazonS3, the second one is send to the end user. If then the version has to be reproduced for a reason, the users sends his key to the administrator who is able to gather the seed and reproduce it.

Item	Attribute
label	Chrome
security	Medium
version	23.0.1271.64
parallel_builds	1
cores_per_build	8
nop_insert	18
mov_lea	50
pre_RA	1
launch_num	100
min_num	20
peak	10-16
min_num_peak	50
update	10
...	...

DNS	timestamp
ec2-12-21-43-43.compute-1.amazonaws.com	1346364501841
ec2-23-21-34-12.compute-1.amazonaws.com	1346365139455
ec2-54-21-12-34.compute-1.amazonaws.com	1346365105630
ec2-73-21-22-54.compute-1.amazonaws.com	1346365146615
ec2-55-21-43-29.compute-1.amazonaws.com	1346365105229
ec2-55-21-76-17.compute-1.amazonaws.com	1346364501841
ec2-32-21-23-23.compute-1.amazonaws.com	1346365152450
ec2-46-21-15-21.compute-1.amazonaws.com	1346420681636
ec2-78-21-53-12.compute-1.amazonaws.com	1346420697714
ec2-33-54-12-53.compute-1.amazonaws.com	1346420692538
ec2-94-23-54-15.compute-1.amazonaws.com	1346420703227
ec2-65-34-12-51.compute-1.amazonaws.com	1346422897189
ec2-54-23-12-78.compute-1.amazonaws.com	1346426796458

Location	ID
acabe3c0-f368-11e1-9f42-96f3dfb052e6.ubuntu.chrome.23_0_1271_64.18501	dfb7361e-f368-11e1-8f88-96f3dfb052e6
e3f61851-f368-11e1-a333-96f3dfb052e6.ubuntu.chrome.23_0_1271_64.18501	f0f4a9de-f368-11e1-9120-96f3dfb052e6

Table 4.1: Domain tables for one task with example values

4.2.3 Amazon SQS

The initial approach was to put the diversified binary into the queue. But Amazon's queue system has a maximum capacity of 64 KB and can only store text which makes this concept impossible. Another problem would have been, if the system runs out of available versions, all users would receive the same binary (instead a random from the reserve) until the system produces enough again.

For this reason, AmazonSQS is only used for the transfer of messages. For the exchange, also Simple Database would have been an option, but SQS provides an important mechanism. When a message is received, it becomes locked, which keeps other computer from processing the message at the same time. This is required when several clients request a download at the same time, also the distribution of seed files to EC2 instances which compile an application need this method.

For each task, two queues are created: one that contains the link to the location of the diversified application and the second to exchange the seed files. The queues use the same syntax as explained in the previous section. To the sequences, the postfixes *seed* and *location* are appended.

4.2.4 Amazon S3

AmazonS3 is used as storage device for files including the Amazon machine images, seed files and the diversified software packages. The top level folders on

AmazonS3 are called buckets and have to be unique because all AmazonS3 users share the same namespace. The number of buckets is limited to 100, but each bucket can contain unlimited objects. To distinguish between the files, the syntax shown from section 4.2.2 is taken.

The App-Store uses two buckets, one for the unique binaries and one for the seed with different access policies. The seed bucket can be accessed from all EC2 units of the Administration view. The unique binaries folder has an additional permission which enables download requests from the end user. But before the client can access a file in this bucket, he must know the download link to make the file accessible from the outside of the cloud. This ensures if an attacker guesses a correct link address (which is very unlikely), he still won't able to download the object.

4.2.5 Amazon EC2

In the administration view, EC2 has four different functions: compiling the applications, creating the seed, host the administration interface and a controller unit that manages the administration side.

4.2.5.1 Administration interface

One AmazonEC2 instance is used to host the admin-website. It provides a simple interface to add, change and delete tasks and AMIs, these changes are saved in the AmazonSDB. Another option to perform this job is using command line tools.

4.2.5.2 Controller

The controller runs on one EC2 instance and has several duties:

- **Manage tasks:** When a task changes, the controller receives a command form the administration instances and updates the according Amazon services. For example, if a new task is created, it adds the queue on AmazonSQS, the bucket on AmazonS3 and launches the required compiler-units.
- **Calculating the required instances:** First, the controller receives the current download numbers from the client (provided per minute) and computes the average from the last hour. Then it calculates the required instances which are needed to build enough software versions for the average amount plus the minimum number specified by the administrator. After this step the controller launches or terminates instances.

- **Updating queues:** Messages on AmazonSQS have a maximum lifetime of 14 days. Therefore, queues are periodically checked, in the case a messages reaches its maximum life span soon, it will be enqueued again.
- **Checking seeds:** Like the computation of the required instances, the right amount of seeds has to be guaranteed.

4.2.5.3 Seed generator

Because the seed generator in its current version creates the files sequentially, an own instances is used to perform this task. The files are uploaded to AmazonS3 and the location is stored on AmazonSQS.

4.2.5.4 Application compiler

The core component of the App-Store is the application compiler unit and works as follows:

1. Load task settings from the database.
2. Receive the seed location from AmazonSQS and load the file from AmazonS3.
3. Build the software.
4. Create keypair and store them on the database.
5. Upload the software and update the seed on AmazonS3.
6. Add the location of the software to AmazonSQS.
7. If the task still exists, continue with 2.

4.3 Client view

The approach for the client side is the following: The end user visits the App-store website and chooses a software with the desired security level and clicks the download button. This action starts a simple web-application that is executed on the local machine in a sandbox and performs the download process. To save the file to the computer the user has to allow a request to save the file on the machine.

The download application retrieves the download link from AmazonSQS and removes it from the queue. After that, the permission on AmazonS3 is changed to

grant access to the chosen software and to begin the download. At the same time, the number of current downloads for this versions is updated on AmazonSDB. When the download is finished, the file will be deleted on AmazonS3, otherwise the file permission changes to unaccessible and the download link will be enqueued again for further requests.

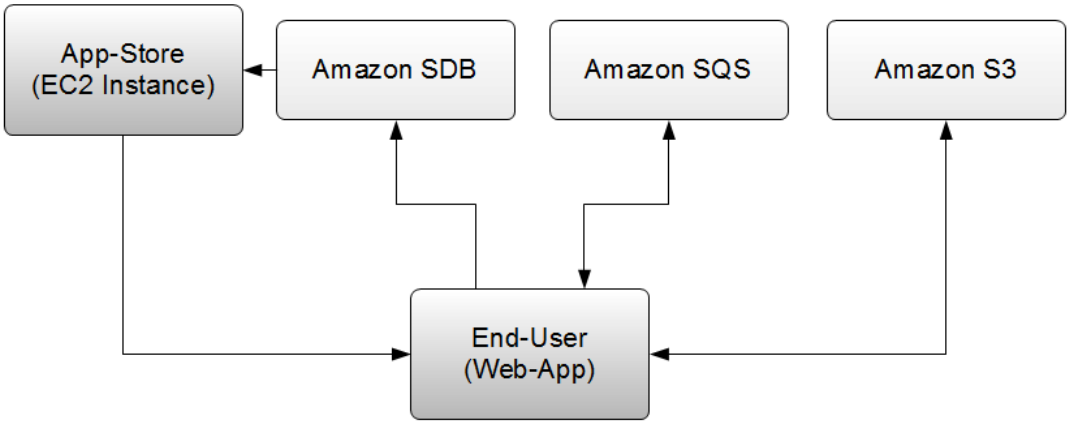

Figure 4.4: Components and their relations in the client view

Bibliography

[BK12] Elaine B. Barker and John M. Kelsey. Sp 800-90a. recommendation for random number generation using deterministic random bit generators. Technical report, Gaithersburg, MD, United States, 2012.

[GPS+03] Daniel Geer, Charles P. Pfleeger, Bruce Schneier, John S. Quarterman, Perry Metzger, Rebecca Bace, and Peter Gutmann. CyberInsecurity: The Cost of Monopoly – How the Dominance of Microsoft's Products Poses a Risk to Security. Computer and Communications Industry Association, 2003.

[Jac11] Todd Jackson. Compile-Time Code Diversification to Raise Attackers' Software Exploit Costs. 2011.

[JGS+12] S. Jajodia, A.K. Ghosh, V.S. Subrahmanian, V. Swarup, C. Wang, and X.S. Wang. *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*. Advances in Information Security Series. Springer London, Limited, 2012.

[JHC+12] Todd Jackson, Andrei Homescu, Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. Diversifying the software stack using randomized nop insertion. In S. Jajodia, A.K. Ghosh, V.S. Subrahmanian, V. Swarup, C. Wang, and X.S. Wang, editors, *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*, Advances in Information Security Series, pages pp 151–173. Springer London, Limited, 2012.

[Kra05] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation, 2005.

[Sha07] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 552–561, 2007.

[TJ] Andrei Homescu Karthikeyan Manivannan Gregor Wagner Andreas Gal Stefan Brunthaler Christian Wimmer Michael Franz Todd Jackson, Babak Salamat. Compiler-generated software diversity.