ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

LST
Laboratory for Software Technology

I Control Your Code

Attack Vectors through the Eyes of Software-based Fault Isolation

Mathias Payer

<mathias.payer@nebelwelt.net>

Motivation

- Current exploits are powerful because
  - Applications run on coarse-grained user-privilege level
    - Every exploit has full user-privileges
  - Local privilege escalation through auxiliary attacks
- Tight security-models limit privileges on both
  - A per-application level and
  - A per-user level
- Idea: each application only has access to the data owned by a specific user that is useful for the application

Fahrplan

- Introduction
- Protection through virtualization
- Attack Vectors
  - Code Injection
  - Return-oriented programming
  - Format String Attacks
  - Arithmetic Overflow
  - Data Attacks
  - x86_64 vs. i386 code
- Demo
- Conclusion

Introduction

- Software security is a challenging problem
  - Both managed and unmanaged languages are prone to attacks
  - Many different forms of attacks exist
- Low-level bugs are omni-present
  - And high-level languages compile down to low-level code
- Hard to eliminate bugs
  - They are hard to find and hard to fix

Introduction

- Programmers rely on too many assumptions
  - That are not necessarily part of the semantics of the programming language, e.g.,
    - Memory layout (little vs. big endian)
    - Type sizes (long is always 4 byte long)
    - Variable placement (layout of structures)

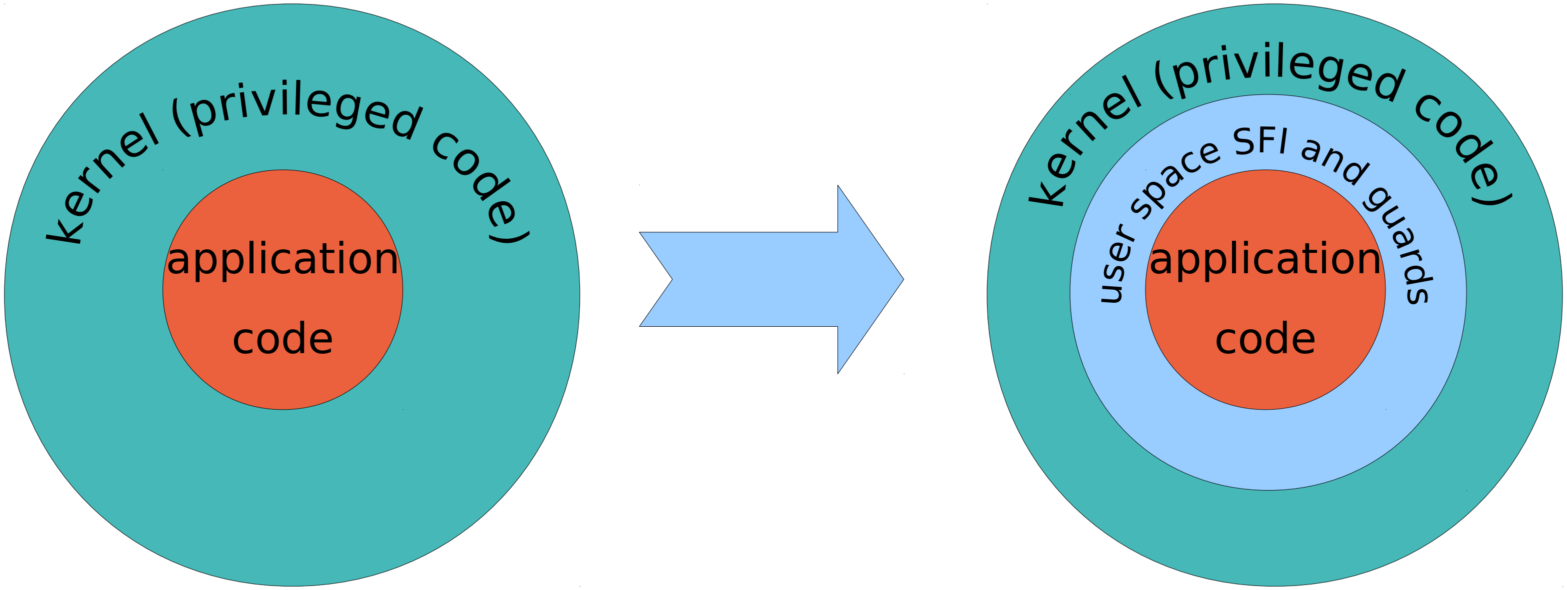
- Goal of this talk:
  - Understand attack vectors and constraints
  - Know how to defend yourself against the attacks
    - Different techniques and security measurements
    - Security analysis
    - Know your assumptions (e.g., language, compiler, architecture)

Fahrplan

- Introduction
- Protection through virtualization
- Attack Vectors
  - Code Injection
  - Return-oriented programming
  - Format String Attacks
  - Arithmetic Overflow
  - Data Attacks
  - x86_64 vs. i386 code
- Demo
- Conclusion

Protection through virtualization

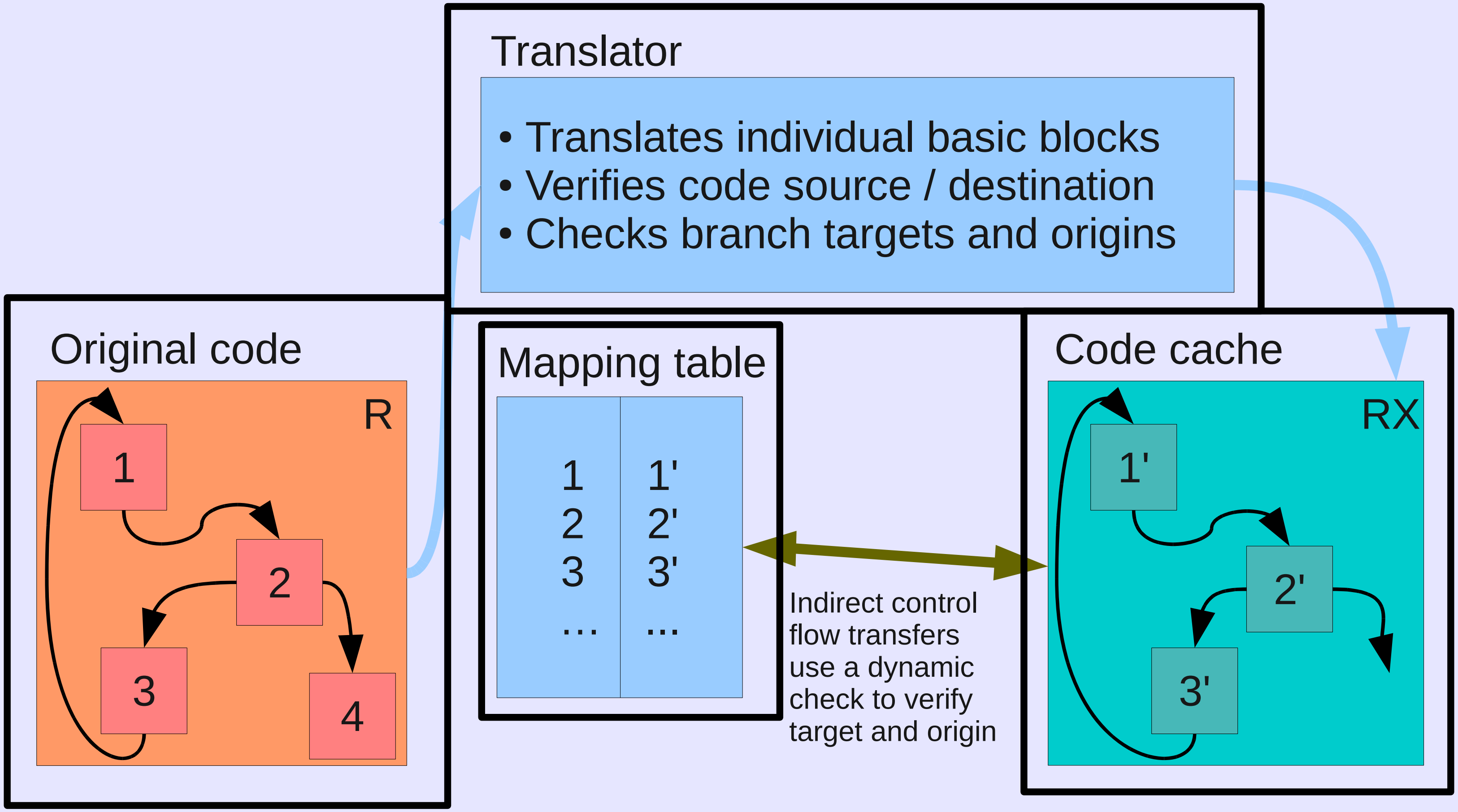
- Like many other problems in CS security can be increased through an additional layer of indirection
- We propose a user-space virtualization system that secures all program code and authorizes all system calls



Protection through virtualization

- Security principles:
  - All code is translated before it is executed. Additional guards are added to the translated code
    - Catches control flow transfers to illegal locations (code injection)
    - Catches illegal control flow transfers (arc attacks)
    - Catches jumps into other instructions
    - Catches switches between i386 and x86_64
  - All system calls are authorized by a policy
    - Catches privilege escalation
    - Catches data bugs that execute unintended system calls

Virtualization in a nutshell



- See: Generating Low-Overhead Dynamic Binary Translators (Mathias Payer, youtube.com/watch?v=VIxaQeAHIxs)

Static security guards

- Check code location
  - Exported in module / object as code region
  - Verify permissions of the page according to the module
- Check target of static control transfers
  - Permission check (through `GOT` – global offset table) for inter-module transfers
  - Verify valid instructions from the beginning of a function to the target of the jump instruction

Dynamic security guards

- Dynamic checks for dynamic control transfers
  - Return instructions, indirect calls, indirect jumps
  - Verify that target is valid and translated
  - Untranslated targets fall back into the static check

- Verify return instructions
  - Validate stack and use a shadow stack

System call authorization

- System calls redirect to an authorization framework
  - Policy based authorization
    - For wide variety of system calls and parameter combinations
  - Authorization functions
    - For redirected system calls
    - Reimplementations of system calls in user space
    - Additional validation of dangerous system calls : `mmap` (overlapping regions); `mprotect` (make code executable); `fork` (new processes); `clone` (new threads)

- System calls are allowed, redirected to the authorization function, or the program is terminated with a security exception

Fahrplan

- Introduction
- Protection through virtualization
- Attack Vectors
  - Code Injection
  - Return-oriented programming
  - Format String Attacks
  - Arithmetic Overflow
  - Data Attacks
  - x86_64 vs. i386 code
- Demo
- Conclusion

Attack vectors

- Attacks redirect control flow
  - New or alternate locations are reached
  - Execution is different from unaltered run

- An attack exploits the fact that the programmer or the runtime system is unable to check
  - the bounds of a buffer or
  - to detect a type overflow or
  - to detect an out-of-bounds access

Code injection

- Injects new executable code into the process image of a running process
  - Into buffer on the stack
  - Into heap-based data structures
- Redirects control flow to the injected code
  - Overwriting the `RIP` (return instruction pointer)
  - Overwriting function pointers, destructors, or data structures of the memory allocator

Code injection: stack-based

- Exploits a missing or incomplete bound check on stack-based buffers

- Exploit uses two steps:
  - Buffer on the stack is filled with machine-code
  - Stack grows downwards and (eventually) overwrites `RIP` with pointer back into the buffer

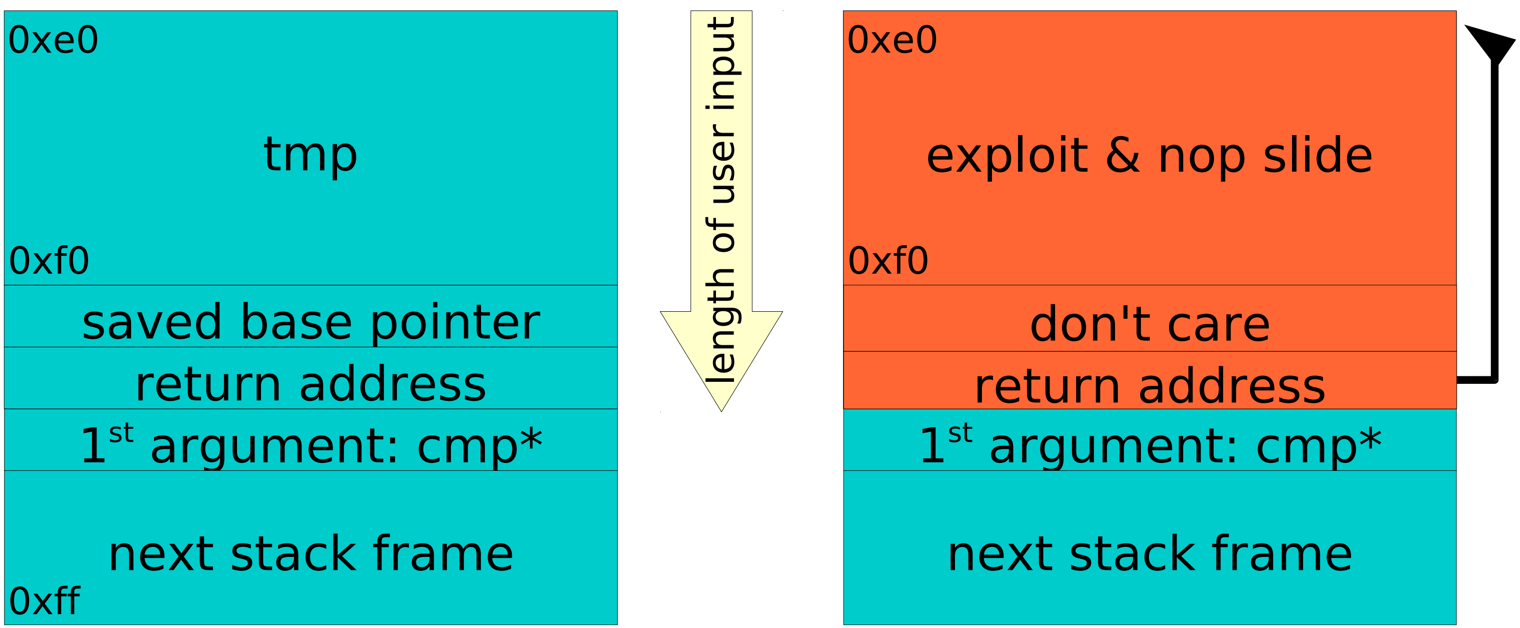
- Constraints
  - Executable stack
  - Missing/faulty bound check
  - `RIP` must not be verified/checked

- See: Smashing the Stack for Fun & Profit (Aleph1, Phrack #49)

Code injection: stack-based

```
int foobar(char* cmp) {
    // assert(strlen(cmp)) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy(tmp, cmp);
    return strcmp(tmp, "foobar");
}
```

No bound checks when data is copied!



Code injection: heap-based

- Exploits a missing or incomplete bound check on heap-based buffers
  - Very similar to stack-based overflows

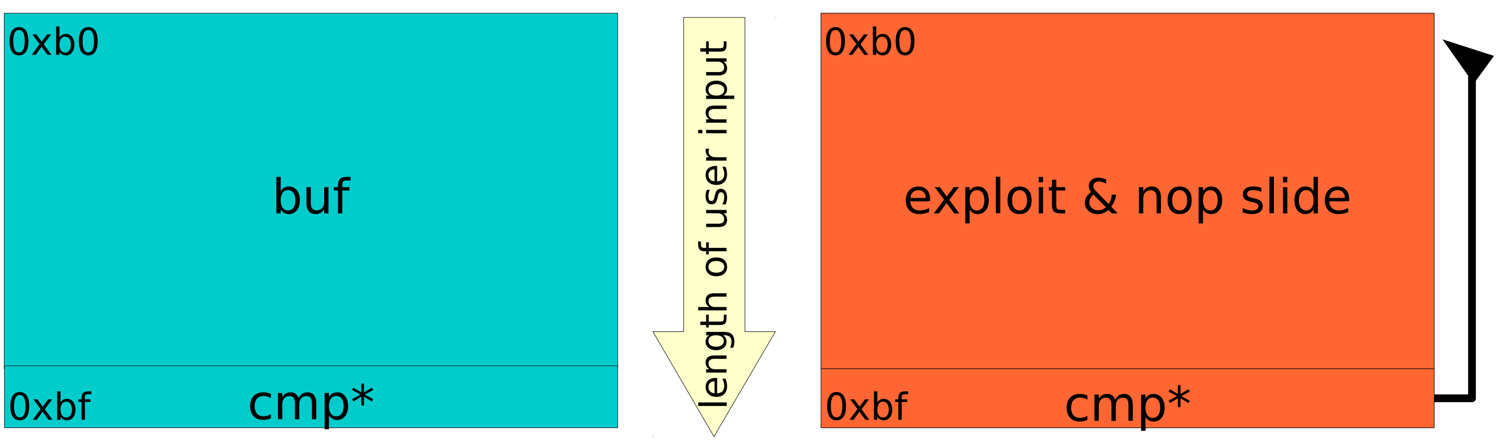
- Exploit uses two steps:
  - Buffer on the heap is filled with machine-code
  - Function pointer, vtable-entry, (GLIBC) destructor, or memory management data-structure altered to redirect control flow

- Constraints
  - Executable heap
  - Missing/faulty bound check
  - Successful redirection of the control flow

Code injection: heap-based

```
typedef struct {
    char buf[MAX_LEN];
    int (*cmp)(char*,char*);
} vstruct;
int is_foobar_heap(vstruct* s, char* cmp) {
    strcpy(s->buf, cmp);
    return s->cmp(s->buf, "foobar");
}
```

No bound checks when data is copied!



Code injection: a tool writers perspect.

- The BT would stop the program when the control flow transfer is detected
  - Before the shellcode is even translated
  - Two exceptions would be triggered
    - Code is (about to be) executed in a non-executable area
    - Function call to an unexported/unknown symbol (heap-based)
    - RIP mismatch (stack-based)

- Use BT to analyze exploits/shellcode
  - Catch new exploits and security holes
  - Use debugging info in application to fix bugs
  - Use BT to audit your own software / test your exploits

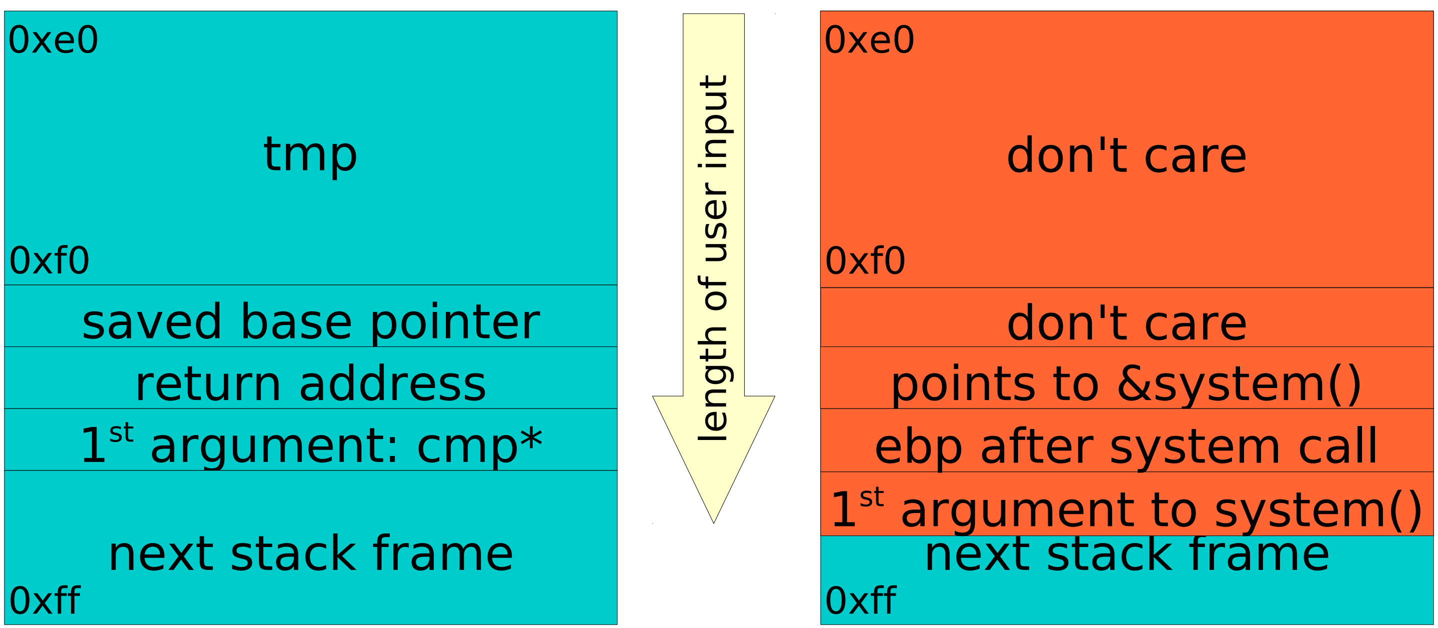
Return-oriented programming

- Exploit already existing code sequences
  - Prepare the stack so that tails of library functions are executed one after another
- Stack-based overflow is used to prepare multiple stack invocation frames
  - Control flow redirected to tails of library functions
  - Tails can be used to execute arbitrary code
- Constraints
  - Missing bound check for the initial stack-based overflow
  - `RIP` must not be checked
- See: Return-Oriented Programming (Shacham, Black Hat'08)

Return-oriented programming

```
int foobar(char* cmp) {
    // assert(strlen(cmp)) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy(tmp, cmp);
    return strcmp(tmp, "foobar");
}
```

No bound checks when data is copied!



ROP: a tool writers perspective

- The BT would stop the program when the control flow transfer is detected
  - Before the function tail or libC function is translated
  - The `execve` system call would be stopped as well
- Real attacks would chain multiple `libC` calls
  - Can be used to inject code into the address space in a legal manner (use `mprotect` to update permissions)

Format string attack

- Exploit the parsing possibilities of the `printf`-family
  - If a user-controlled string is passed to a `printf` function

- A combination of `%x` and `%n` in strings that are passed to `printf` unfiltered result in random memory reads and random memory writes
  - Careful preparation of the input is needed

- The format string must be allowed to contain `%n` and, e.g., `%x` to write to memory
  - Random writes can be used to redirect the control flow by overwriting, e.g., the `RIP`, destructors, or the vtable

Format string attack

- `printf` examples:
  - `printf("val: %d, ptr: %p, str: %s\n", a, &a, str);`
    - `// value: 12, ptr: 0x7ffffffe27c, str: foobar`

- Interesting `printf` features
  - `%NN$x` – use the NN-th parameter (out of order access)
  - `%MMx` – print the parameter using MM bytes
  - `%NN$MMx` – print the NN-th parameter using MM bytes
  - `%hn` – write the amount of printed characters to the given parameter
  - `%KK$hn` – write the amount of chars to KK-th parameter

- General idea:
  - Combine these features for random writes to memory

Format string attack

```
void foo(char* cmp) {
	char text[1024];
	strcpy(text, cmp);
	printf(text); // correct: printf("%s", text);
}
```

- Use references to our string to retrieve pointers
  - **`\x7c\xd3\xff\xff\x7e\xd3\xff\xff%12$2043x.%12$hn%11$32102x%11$hn`**
  - Writes `0x0804` to `0xffffd37e` and `0x856a` to `0xffffd37c`
    - First 8 bytes of the string contain 2 pointers to half words
    - `%12$2043x` prints 2043 bytes (increases the # of printed bytes)
    - `%12$hn` writes a half word with the # of printed bytes to the first address
    - `%11$32102x` writes 32102 more bytes
    - `%11$hn` writes the second half word to memory
    - This redirects the return instruction pointer to our special function

Format string: a tool writers perspect.

- BT stops the program when the control flow is redirected
  - Change of `RIP` to new function
  - System call guard checks arguments of system calls
  - Policy violations are detected and the program is stopped

- Random writes to memory only detectable with full memory tracking

Arithmetic overflow

- Exploit overflows in data types
  - Sometimes the bounds are checked before an arithmetic operation
- Data types are of a specific length, arithmetic operations can cause overflows or underflows
  - Dangerous (unchecked) values passed to functions
- Constraints
  - Lax or implicit type conversions
  - Sign errors, rounding errors, type overflows, and wrong pointer arithmetic

Arithmetic overflow

```
/* found in: OpenSSH 3.3 */
nresp = packet_get_int();
if (nresp > 0) {
 response = xmalloc(nresp*sizeof(char*));
 for (i = 0; i < nresp; i++)
  response[i] = packet_get_string(NULL);
}
```

- Seems correct on first look
  - But pass `0x40000000` as len parameter
    - `sizeof(int)=4`
    - `0x40'00'00'00*4 = 0x1'00'00'00'00 = 0x00'00'00'00`
  - `xmalloc()` suceeds
    - Following loop overwrites (large) parts of memory

Arith. ovfl: a tool writers perspective

- Often used to overwrite memory and prepare secondary attack
  - Or can be used to inject code (similar to buffer overflow)
- BT detects the control flow transfer to illegal code
  - System call authorization protects from system call only attacks
- Food for thought: Use BT to analyze EFLAGS

Data attack

- Exploits a missing or faulty bound check
  - Writes data to an user-controlled address
  - Almost as much fun as format-string exploits
- Results in a random write to memory
  - Position and value often only partially checked
    - Use, e.g., integer overflow or combine with other attack

```
void foo(int pos, int value, int* data) {
      data[pos] = value;
}
```

Data attack: a tool writers perspective

```
void foo(int pos, int value, int* data) {
    data[pos] = value;
}
```

```
movl 0x8(%ebp),%eax  ; pos
shll $0x2,%eax       ; pos * 4
addl 0x10(%ebp),%eax ; data + pos*4
movl 0xc(%ebp),%edx  ; value
movl %edx,(%eax)     ; data[pos]=val.
```

- Random (4b) write to memory
  - Relative to position of data array (static?)
  - Constraint: pos and value are user controlled

- Hard to detect, BT stops illegal control flow transfers or illegal system calls

Mixing x86_64 and i386 code

- Modern kernels support both x86_64 and i386 code in parallel
  - Code can even be mixed in one program

- System call authorization tools and static verifiers work on the assumption that only one form of machine code is used
  - System calls have different numbers in 32bit and 64bit mode
  - Checkers can be tricked to allow dangerous system calls

- Only a dynamic runtime security system that is 64bit aware can guard against these threats

- See: Bypassing syscall filtering technologies (Chris Evans)

Mixing code: a tool writers perspective

- System calls mixup even worse for `seccomp`-based sandboxes
  - Seccomp allows `exit`, `read`, `write`, `sigreturn`
  - Mixe-up allows `stat` or `chmod`
- BT detects long jump that switches between i386 and x86_64 mode
  - Syscall tables switched accordingly

Demos

- This is what you've been waiting for

- Which exploit do you want to see?
  - Heap-based code injection
  - Return-oriented programming
  - Format string attack

- Please vote!

Code injection: heap-based (DEMO)

- Open file, read data from file and execute `is_foobar_heap`
  - Compare function is overwritten depending on file input

- The BT would stop the program when the control flow transfer to the heap is detected
  - Before the shellcode is even translated
  - Two exceptions would be triggered
    - Code is (about to be) executed in an non-executable area
    - Function call to an unexported/unknown symbol

- Without security enabled BT translates and disassembles exploit code
  - See exploit dump

Return-oriented programming (DEMO)

- Simple stack-based overflow used to prepare a single stack frame
  - Call to system with `/bin/sh` as parameter
  - Original program fails after return from system

- The BT would abort the program when the change of the `RIP` is detected
  - The `RIP` now points to a non-exported symbol
  - The `execve` system call would be stopped as well

- BT without security translates and executes `execve`
  - Log shows last entry with a system call

Format string attack (DEMO)

- Attack combines two half-word writes
  - Only RIP overwritten
  - Real exploit would overwrite multiple words to prepare second stage of attack
- The BT would abort the program when the change of the `RIP` is detected
  - The `RIP` now points to a non-exported symbol
  - The `execve` system call would be stopped as well
- BT without security translates and executes `execve`
  - Log shows last entry with a system call

Conclusion

- User-space BT contains security problems
  - Security violations detected and program terminated
  - User-space, fine-grained, per-process, per-user model of security
- Virtualization used to analyze threats
  - Analyze malicious payload
  - Observe control transfers and locations of break-ins
- fastBT supports the full ia32 ISA; x86_64 almost complete; no kernel modification necessary
  - All forms of system calls checked and authorized
- Source & demos: http://nebelwelt.net/fastBT