Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Laboratory for Software Technology

secuBT
–
Hacking the Hackers with User-Space Virtualization

Mathias Payer
<mathias.payer@inf.ethz.ch>

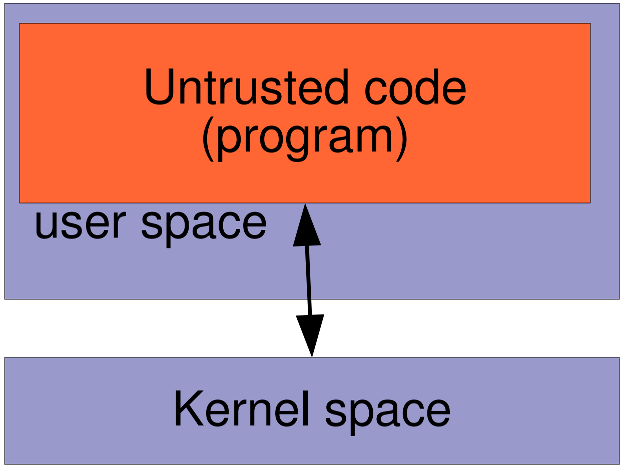
Motivation

Virtualizing and encapsulating running programs is important:

Sandboxing for server processes to guard against unknown software vulnerabilities

Execution of untrusted code

Offers different security contexts per user



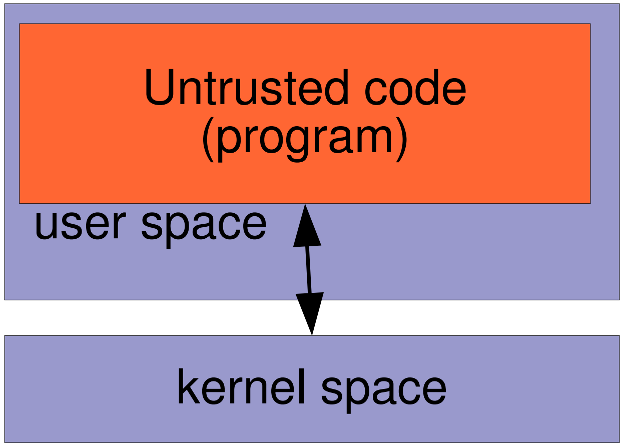
Programs can execute any system call

No custom-tailored selection

Security vulnerabilities can be used to execute unintended system calls

These are not typical for the application

Patches are a ***reactive*** form of security



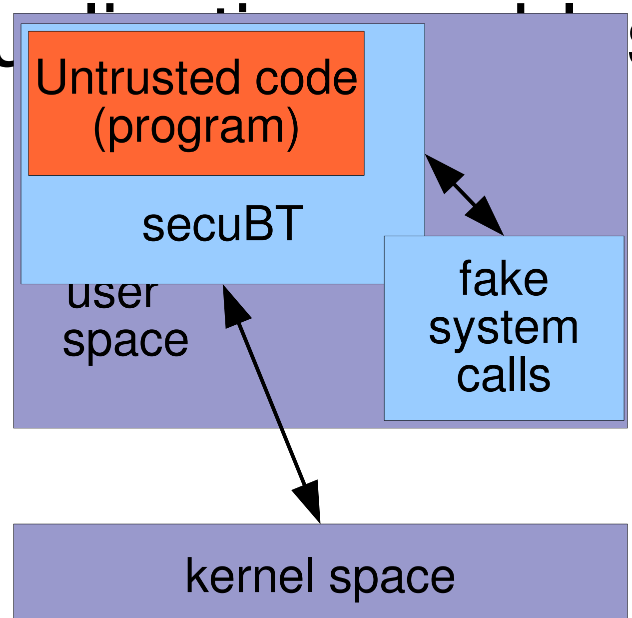
User-space virtualization encapsulates a running program

Executed code is checked & validated

Code can be wrapped or modified

System calls are validated before execution

User-space virtu… s a ***proactive*** form of security



- User-Space Virtualization is implemented through Dynamic Binary Translation
  - Binary Translation as the art of adding, removing or replacing individual instructions
  - Control over all user-code instructions

secuBT implements a User-Space Sandbox

Dynamic BT used for the virtualization layer

Privilege separation in user-space to guard BT

System Call Interposition Framework

Checks and validates all System Calls

Ensures that the program cannot break out of the virtualization layer

Fahrplan

- Introduction
- Design and Implementation
  - User-Space Virtualization through BT
  - Basic Translator
  - Optimizations
  - Security Hardening
  - System Call Interposition Framework
- Performance & Demonstration
- Conclusion

Dynamic BT

Binary Translation (BT) as a program instrumentation tool

Static vs. dynamic BT

BT unit translates code before it is executed

Checks and validates instructions based on translation tables

Two levels of code execution:

'Privileged' code of the BT library

Translated and cached user code

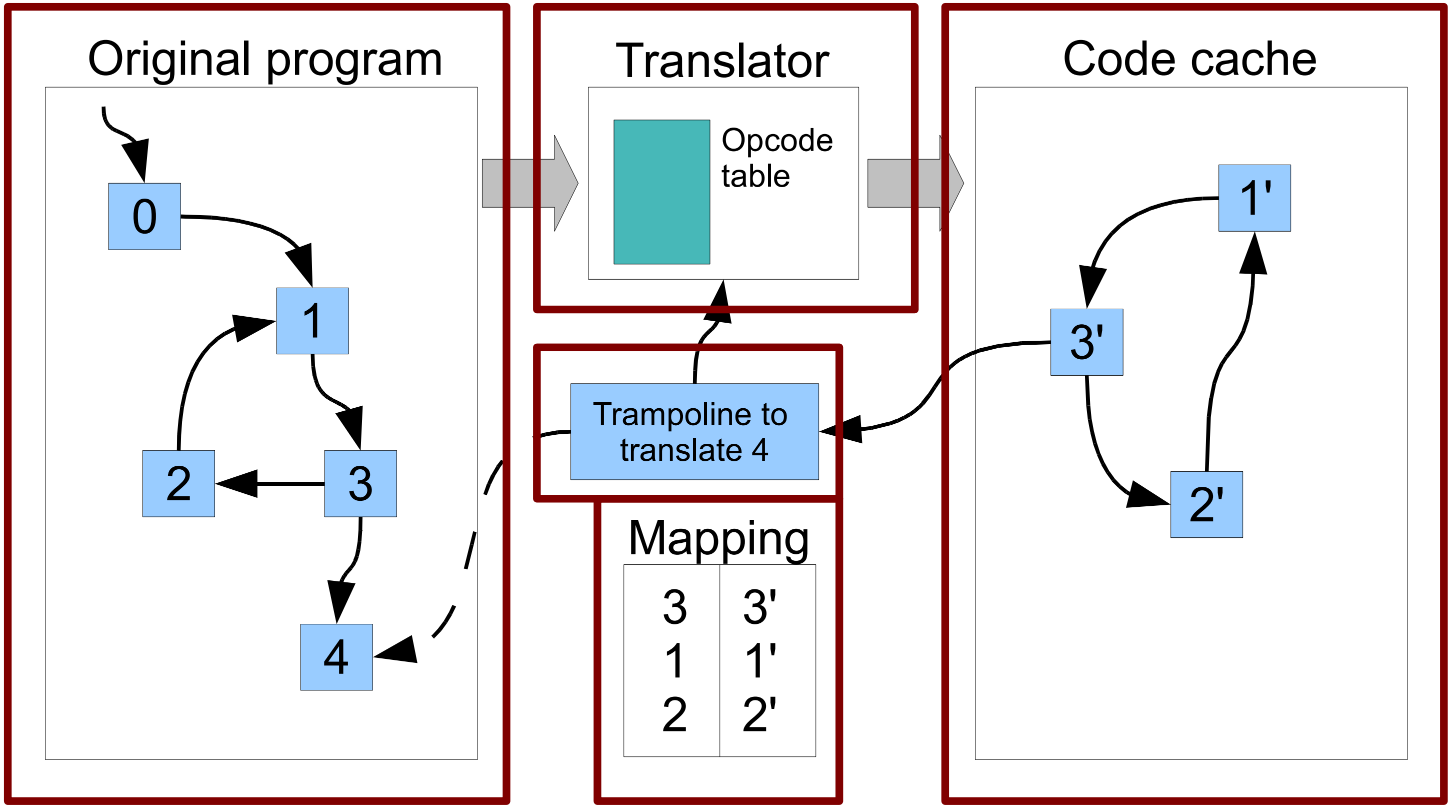
Using BT the program is encapsulated in an additional protection layer

All instructions are checked

All (direct & indirect) jump targets are verified

All system calls are verified

BT in a nutshell:



Fahrplan

- Introduction
- Design and Implementation
- User-Space Virtualization through BT
- Basic Translator
- Optimizations
- Security Hardening
- System Call Interposition Framework
- Performance & Demonstration
- Conclusion

Translation efficiency

Fast table-based translation process

Code cache

Efficiency of generated code

Master control transfers

Indirect jumps

Indirect calls

Function returns

Optimization: Translation Efficiency

Table-based iterator

No global state or IR needed

Instructions decoded according to information in the translation tables

Local peephole optimizations like inlining still possible

Based on intermediate representation (IR)

IR transformation and global state needed

Optimizations based on IR rewriting

IR transformed back to machine code

Indirect control flow transfers are expensive

Runtime lookup & patching

Indirect control transfer replaced by software trap!

```
 Calculate target address from original instr.
 Execute software trap
Lookup target (translated?)
Fix return address and redirect to target
 Ensure that no ptr. to code cache leak
```

1 ins

Can we avoid that? Or lower the cost?

Optimization: Efficient Code

Be clever about the code secuBT generates!

Instruction encodings are manifold:

Choose best fitting optimization

Translate different indirect calls:

'Static': `call *(fixed_location)`

Use a static prediction

'Dynamic': `call *(%reg)`

Use inlined, fast dispatch

Combination possible as well

Static ind. call: `call *(fixed_location)`

```
pushl src_addr
jmp *xx(ind_target)
```

```
pushl src_addr                              (1)
cmpl $cached_target, *xx(i_trgt)            (2)
je $trans_target
pushl *xx(ind_target)                       (3)
pushl $tld
pushl $addr_of_cached_target
call fix_ind_call_predict
```

1. Push original src IP
2. Compare actual target w/ cached target & branch if prediction ok
3. Recover if there is a misprediction

Optimization: Efficient Code

Dynamic ind. call: `call *(reg)`

```
pushl src_addr
jmp *(reg)
```

```
pushl src_addr, *(reg), %ebx, %ecx
movl 12(%esp), %ebx          # load target
movl %ebx, %ecx              # duplicate ip
andl HASH_PATTERN, %ebx      # hash fct
cmpl hashtlb(0, %ebx, 8), %ecx    # check
jne nohit
movl hashtlb+4(0, %ebx, 8), %ebx # load trgt
movl %ebx, (tld->ind_jmp_targt)
popl %ecx, %ebx              # epilogue
leal 4(%esp), %esp           # readjust stack
jmp *(tld->ind_jmp_targt) # jmp to trans.trgt
nohit: use ind_jump to recover
```

Optimization: Efficient Code

Many more optimizations available:

Return instructions

Shadow stack or fast dispatch

Indirect jumps

Jumptable optimization

Prediction, and fast dispatch

Function inlining

Complete or partial function inlining

Optimizations bring competitive performance!

Fahrplan

- Introduction
- Design and Implementation
- User-Space Virtualization through BT
- Basic Translator
- Optimizations
- Security Hardening
- System Call Interposition Framework
- Performance & Demonstration
- Conclusion

BT enables additional security checks:

- Enforce NX-bit
- Check ELF headers, regions, and rights
- Protecting internal data structures (`mprotect`)
- Check and verify (valid) return addresses
- Checking & verifying indirect control transfers

Security: NX-bit

Enforcing the NX-bit (1 bit / page)

IA32 does not enforce the executable bit

Only regions that are marked executable are allowed to contain code

If code branches to a NX-region the program is terminated

The translator checks for every new block if the source code is from an executable region

Security: ELF headers

Checking ELF headers, regions, and rights

Check call instructions to only call exported functions

Check jump instructions to stay inside individual modules

Enforce defined access rights on ELF regions, not on coarse-grained pages

Protecting internal data structures

Use `mprotect` calls to (write-)protect all internal data structures

Remove protection when switching to the VM

Reinstantiate protection when returning to user-code

Write-protect all translated user-code regions and libraries

Trade-off: Probabilistic to explicit protection through additional `mprotect` calls

Check and verify (valid) return addresses

Match return addresses with addresses on a shadow stack hidden from user code

Check and verify indirect control transfers

Validate target of indirect control transfers

Indirect calls (e.g., function pointers)

(Valid) return addresses

Indirect jumps (e.g., switch tables)

If target is not valid code, not in a code region, or the control transfer is illegal (e.g. jump into a different library) terminate the program

Fahrplan

System Call Interposition

System calls through `sysenter` & `int 80` redirected to validation function

Depending on the system call and the arguments the system call is:

Allowed and executed

Disallowed and the program terminated

Redirected to some user-space function

Validation based on checker functions on a per system call basis

Redirect system call to user-space function:

```
const authorize_syscl_fptr_t
   authorize_syscl_table[] = {
 ...
 intercept_getid, // __NR_getuid32      199
 intercept_getid, // __NR_getgid32      200
 intercept_getid, // __NR_geteuid32     201
 intercept_getid, // __NR_getegid32     202
 allow_syscall,   // __NR_setreuid32    203
 allow_syscall,   // __NR_setregid32    204
 ...
};
```

Implement function that handles system call in user space:

```
int intercept_getid(int syscall_nr,int
arg1,int arg2,int arg3,int arg4,int arg5,
int arg6,int is_sysenter,int *retval)
{
    // yes, we simulate root ;)
    *retval = 0;
    // 0 - return fake value
    // 1 - allow system call
    return 0;
}
```

Demo time!

Fahrplan

- Used SPEC CPU2006 benchmarks to measure translation overhead
- Three different configurations:
  - Overhead of BT alone
  - BT, and syscall authorization overhead
  - BT, syscall auth., and `mprotect` overhead
- System:
  - Ubuntu 9.04, GCC 4.3.3
  - E6850 Core2Duo CPU @ 3.00GHz, 2GB RAM

Evaluation: Overhead

Benchmark	BT alone	secuBT	sBT+mprot
400.perlbench	66.87%	67.70%	72.22%
401.bzip2	4.34%	3.89%	4.19%
403.gcc	32.20%	31.97%	84.81%
429.mcf	0.25%	0.00%	0.25%
458.sjeng	36.04%	35.90%	35.76%
464.h264ref	8.19%	10.21%	10.21%
483.xalancbmk	30.19%	29.38%	32.35%
416.gamess	-3.50%	-2.80%	-2.10%
Average	**6.93%**	**7.44%**	**9.36%**

Compared to uninstrumented run

Selection of benchmarks shown

Overhead is ***low*** and ***tolerable***

What protection does ***secuBT*** offer?

Heap and stack based overflows

As soon as code is 'to be' executed

Return to libc attacks

If you deny all unneeded system calls

Overwriting the return instruction pointer

If you use save shadow stack

```
void myfunc(int argc, char *argv[])
{
  char buf[4];
  sprintf(buf, "%s", argv[1]);
}

int main(int argc, char *argv[])
{
  myfunc(argc, argv);
  return 0;
}
```

Demo time: exploit

```
#define SHELL 0xffffdfe9
#define SYSTEM 0x804836c
 ...
 char shell[] = "SHELL=/bin/tcsh";
 char *env[3] = { ldpreload, shell, 0 };
 for (i=0; i<RIPOFFSET; i++) buf[i]='A';
 *(int*)(buf+RIPOFFSET) = SYSTEM;
 *(int*)(buf+RETADDR) = 0x10c0ffee;
 *(int*)(buf+TARGETEXEC) = SHELL;
 buf[TARGETEXEC+4]=0;
 execle("./bof", "./bof",
    (const char*)&buf, (char *)0, env);
```

Demo time!

Fahrplan

- Introduction
- Design and Implementation
- User-Space Virtualization through BT
- Basic Translator
- Optimizations
- Security Hardening
- System Call Interposition Framework
- Performance & Demonstration
- Conclusion

Conclusion

Use secuBT for proactive security

Contain and detect memory corruption

Additional protection without recompilation

Uses dynamic BT to support full IA32 ISA without kernel modifications

Intercept interactions of the program with the kernel, e.g., system calls, signals

Questions?

Source code & project:

- http://nebelwelt.net/projects/secuBT

Thanks to

- The albtraum team
- My colleagues for comments & reviews
- Marcel Wirth, Peter Suter, Stephan Classen, and Antonio Barresi for code contributions

ptrace vs. User-Space Virtualization

- ptrace needs kernel support and stops traced programs in kernel space (on signals)
  - Must trust code in kernel
  - Coarse grained checking, not per-instruction
  - High overhead per system call, low overhead for user-space parts
- secuBT runs completely in user-space
  - Small trusted code base
  - Fine grained validation and checking
  - Additional hardening (NX, Stack check, ...)
  - BT and translation overhead (-3.5% ... 10%)

What about eflags?

Static ind. call: `call *(fixed_location)`

```
pushl src_addr
jmp *xx(ind_target)
```

```
pushl src_addr
pushfl
cmpl $cached_target, *xx(ind_target)
jne $nohit
popfl
jmp $trans_target
$nohit popfl
pushl *xx(ind_target)
pushl $tld
pushl $addr_of_cached_target
call fix_ind_call_predict
```