Kimchi: A Binary Rewriting Defense against Format String Attack

loafers

DOUL Infotech

Power of Community(POC) 2006
Nov. 2006

Security Patch by Binary Rewriting is Required

In case

- the majority of distributed binary programs are still built without security defense mechanism
- we cannot rebuild the binary program from the patched source code
- we cannot get the patched binary program from the vendor in a timely manner
- a malicious developer might make security holes intentionally

Previous research into binary rewriting for security patch

- [Prasad, 2003]: A binary rewriting defense against stack-based buffer overflow attacks

Research Objective

We propose a security patch tool, Kimchi

- modifies binary programs of Linux/IA32
- built without any source code information
- even if the `libc` library is statically linked to them, or
- they do not use the frame pointer register
- to defend against format string attacks in runtime.

Unsafe printf function call

myecho.c: echo C program

```
int main(int argc, char *argv[])
{
    int i = 10;
    if (argc > 1)
        printf(argv[1]);
    printf("\n");
}
```

Nothing wrong happened

```
$ ./myecho 'hello world'
hello world
```

What happened here?

```
$ ./myecho '%x %x %x %9$d %12$d %62$s'
0 bfe04cb8 80483d6 10 2 USER=jhyou
```

Why did this happen?

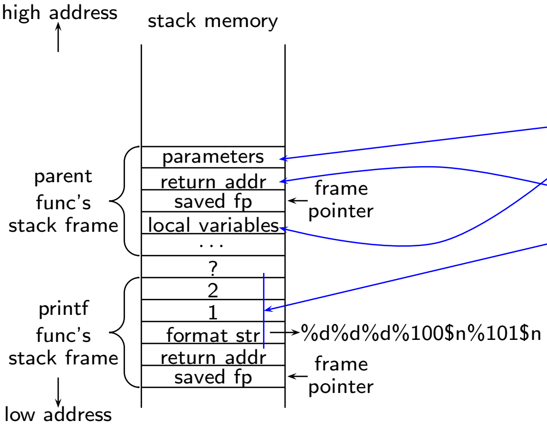
`printf("%x %x %x %9$d %12$d %62$s");` leads the unexpected behaviour!

The safe code

`printf("%s", argv[1]);` instead of `printf(argv[1]);`

How Harmful Format String Vulnerability is

`printf("%d%d%d%100$n%101$n",1,2)` and format string attack



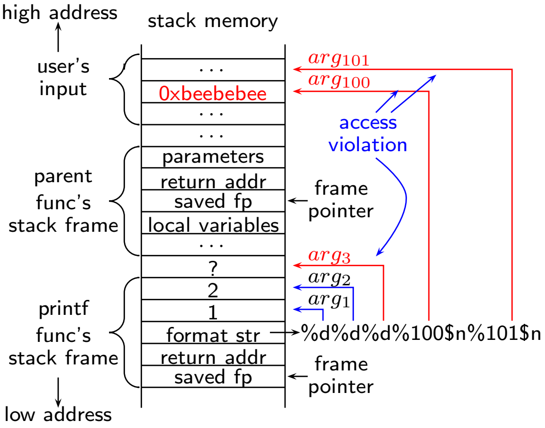
Stack frames for running function on stack memory.

```
void parent(int parameters)
{
    int localvariables;
    printf("%d%d%d%100$n%101$n",1,2);
    return;
}

int printf(char *format, ...)
{
    ...
}
```

How Harmful Format String Vulnerability is

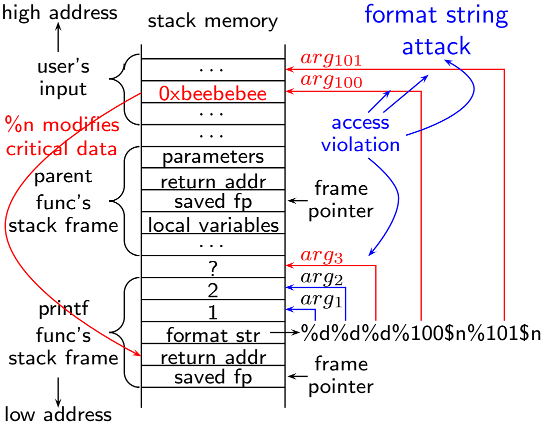
`printf("%d%d%d%100$n%101$n",1,2)` and format string attack



- Accesses of arg_3, arg_{100} and arg_{101} are violations.
- However, printf does not check it.
- This can make security hole!

How Harmful Format String Vulnerability is

`printf("%d%d%d%100$n%101$n",1,2)` and format string attack



- %n stores the number of written characters. `printf("hello%n", &len)` stores 5 into `len`.
- %100$n changes the return address of printf to disorder program's control flow.
- Using %n, attacker can execute arbitrary codes!

Cause and Solution of Format String Vulnerability

Causes of format string vulnerability

- programmer's unsafe coding:
  `printf`'s format string contains user modifiable input string.
- unsafe `printf` implementation in standard library:
  no checking of access validity of format directives

Solutions

- re-code all format strings not to conatin any user input strings.
- improve `printf` to check the safety of format string at runtime.

Historical Review

From when and how many

- Since Tymm Twillman's report to bugtraq in 1999
- 30∼40 public reports of format string vulnerability per year

Case Study

- proftpd-1.2.0pre6/src/main.c:782, the first, 1999

```
snprintf(Argv[0], maxlen, statbuf);
```

  instead of

```
snprintf(Argv[0], maxlen, "%s", statbuf);
```

- bind-4.9.5/named/ns_forw.c:353, CVE-2001-0013, 2001

```
syslog(LOG_INFO, buf);
```

  instead of

```
syslog(LOG_INFO, "%s", buf);
```

Researches into Defense against Format String Attack I

Source Code Level

- [Shankar, 2001]:
  at pre-compile time,
  detecting format string vulnerabilities using type qualifier
- FormatGuard:
  at compile time,
  replacing automatically `printf` function calls in source program with the calls to safe `__protected_printf`
- CCured: a dialect of C Language
  at compile time,
  providing safer *vararg* macro functions

Researches into Defense against Format String Attack II

Binary Level (Without Special Source Code Information)

- libformat, libsafe:
  at program loading time,
  linking to the protected version of `printf` instead of the original in the standard library.
- TaintCheck:
  at program running time,
  Tracing user-input data propagations in the monitored program, and checking whether the user-input is included in the format string.

※ Kimchi's binary rewriting is done at pre-execution time.
Kimchi protects binary programs WIHOUT any special source code information.

Weakness of Previous Binary Level Defense Methods against Format String Attack

libformat and libsafe are applicable ONLY to binary programs

- dynamically linking `libc.so`, the standard C library
- compiled to use the frame pointer register in case of libsafe

TaintCheck SLOWS the traced program execution by a factor 1.5 to 40:

- it runs a binary program in traced mode like a debugger,
- monitors all binary code and tracks the propagation of user input data

Generic binary level defenses NOT SPECIALIZED to format string vul.

- do not prevent invalid argument accesses of printf.
- make the exploit difficult to succeed but NOT IMPOSSIBLE.

Code Pattern of Function Call Passing Parameters

C code of printf call with paramters

```
printf("%d%d%d%100$n", 1, 2);
```

Basic binary code pattern generated by an IA32 compiler

```
subl  $4, %esp    ; for 16 byte memory alignment of parameters
pushl $2          ; param3| push parameters into the stack
pushl $1          ; param2| format argument range: 2 * 4 = 8 byte
pushl $.FMT       ; param1
call  printf      ; call function
addl  $16, %esp   ; remove memory for parameters

.FMT: .string "%d%d%d%100$n"  ; stored in the data segment
```

- The optimized code can be different and complicated.
- Kimchi can detect only the basic pattern currently.

Read-only Format String is SAFE!

printf call with Constant Format String

```
    C code: printf("%d %d %d %100$n", 1, 2);
Binary code:
 804836e: 83 ec 04        sub  $0x4,%esp
 8048371: 6a 02           push $0x2
 8048373: 6a 01           push $0x1
 8048375: 68 88 84 04 08  push $0x8048488
 804837a: e8 31 ff ff ff  call 80482b0 <printf>
 804837f: 83 c4 10        add  $0x10,%esp
```

- Read-only constant string cannot be modified, so not vulnerable basically
- Kimchi does not patch `printf` call with constant format string

ELF binary file information

```
foo:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
 13 .rodata       00000015  08048480  08048480  00000480  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA

Contents of section .rodata:
 8048480 03000000 01000200 25642564 25642531  ........%d%d%d%1
 8048490 3030246e 00                          00$n.
```

Rewriting of printf Call WITHOUT Extra Arguments

Original Binary Code

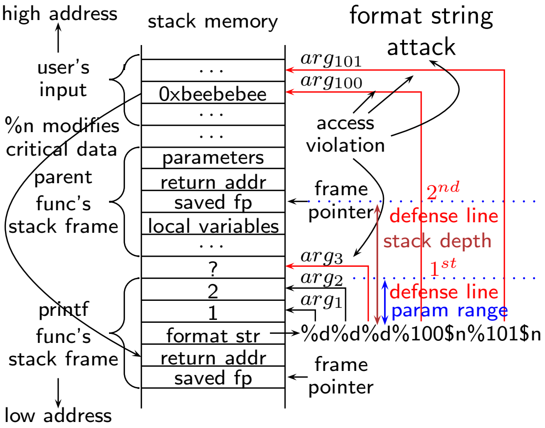
```
main:
 ...
 subl   $12, %esp ; for 16 byte alignment
 movl   $12(%ebp), %eax
 addl   $4, %eax   ; %eax = &argv[1]
 pushl  (%eax)     ; format string arg.
 call   printf     ; printf(argv[1])
 addl   $16, %esp ; remove arguments
 ...
```

- printf call without extra arguments: printf(string);
- call printf is replaced with safe_printf_noarg which calls printf("%s", string) instead of printf(string) to remove the vulnerability.

Rewritten Binary Code

```
main:
 ...
 subl   $12, %esp
 movl   $12(%ebp), %eax
 addl   $4, %eax
 pushl  (%eax)
 call   safe_printf_noarg
 addl   $16, %esp
 ...
safe_printf_noarg:     ; INSERTED CODES
 movl $4(%esp), %eax
 subl $4, %esp
 pushl %eax       ; format_str arg.
 pushl $.FMT      ; "%s"
 call  printf     ; printf("%s",format_str)
 addl  $12, %esp
 ret
.FMT: .string "%s"
```

Defense Idea of safe_printf with Extra Arguments



- Kimchi replaces binary codes to call printf with ones to call safe_printf
- safe_printf protects from accessing over "1st or 2nd defense line"
- stack depth as the range of parameters is passed to safe_printf when Kimchi can not determine the parameter range.
- The same defense method is applied to fprintf, sprintf, snprintf, syslog, warn, err, ...

Concept of replacing call to printf with safe_printf

Original Code

```
void foo()
{
  int a, b, c;
  printf("%d%d%d%100$n", 1, 2);
}
```

- safe_printf checks the argument access over the parameter range.
- if safe, calls original printf,
- otherwise, runs response routine against the attack.

Replaced Code in Concept

```
void foo()
{
  int a, b, c;
  safe_printf(20, "%d%d%d%100$n", 1, 2);
} /* stack depth = 20, exact param range = 8 */
/* inserted code */
int safe_printf(int paramrange,char* format,...)
{
  if (is_safe(format, paramrange)) {
    va_start(ap, format);
    rc = vprintf(format, ap);
    va_end(ap);
    return rc;
 } else {
    /* format string attack is detected */
 }
}
```

Code Pattern of Printf Call with Extra Arguments

C code of printf call with extra arguments

```
printf("%d%d%d%100$n", 1, 2);
```

Basic binary code pattern generated by an IA32 compiler

```
subl  $4, %esp    ; for 16 byte memory alignment of parameters
pushl $2          ; param3| push parameters into the stack
pushl $1          ; param2| format argument range: 2 * 4 = 8 byte
pushl $.FMT       ; param1
call  printf      ; call function
addl  $16, %esp   ; remove memory for parameters

.FMT: .string "%d%d%d%100$n"  ; stored in the data segment
```

- The optimized code can be different and complicated.
- Kimchi can detect only the basic pattern currently.

Rewriting of printf call with DETERMINED arguments

Original Binary Code

```
.FMT: .string "%d%d%d%100$n"
foo:
  ...
  subl  $4, %esp
  pushl $2               ; format parameter
  pushl $1               ; range: 4 + 4 = 8
  pushl $.FMT
  call  printf           ; printf(.FMT,1,2)
  addl  $16, %esp
  ...
```

- call printf is replaced with safe_printf_sp_8 corresponding the parameter range value(8).
- safe_printf_sp_8 calls safe_printf passing the parameter range value.

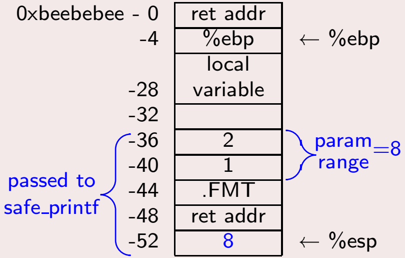
Rewritten Binary Code

```
.FMT: .string "%d%d%d%100$n"
foo:
  ...
  subl  $4, %esp
  pushl $2
  pushl $1
  pushl $.FMT
  call  safe_printf_sp_8
  addl  $16, %esp
  ...

safe_printf_sp_8:    ; INSERTED CODES
  pushl $8            ; param range = 8
  call  safe_printf   ; safe_printf(8,
  addl  $4, %esp      ;  retaddr, fmt,...);
  ret
safe_printf:
  ...
```

Rewriting of printf call with DETERMINED arguments

Change of the Stack



- safe_printf_sp_8(.FMT, 1, 2) calls safe_printf(8, retaddr, .FMT, 1, 2).

Rewritten Binary Code

```
.FMT: .string "%d%d%d%100$n"
foo:
   ...
   subl  $4, %esp
   pushl $2
   pushl $1
   pushl $.FMT
   call  safe_printf_sp_8
   addl  $16, %esp
   ...

safe_printf_sp_8:    ; INSERTED CODES
   pushl $8           ; param range = 8
   call  safe_printf  ; safe_printf(8,
   addl  $4, %esp     ;  retaddr, fmt,...);
   ret
safe_printf:
   ...
```

Passing Stack Depth In a Function USING Frame Pointer

Original Binary Code for foo()

```
.FMT: .string "%d%d%d%100$n"
foo:
  pushl %ebp          ; setup frame pointer
  movl  %esp, %ebp    ;
  subl  $24, %esp     ; alloc local var mem
  subl  $4, %esp      ; typical pattern of
  pushl $2            ; function call
  pushl $1            ;
  pushl $.FMT         ; printf(.L0,1,2);
  call  printf        ;
  addl  $16, %esp     ;
  leave               ; reset frame pointer
  ret                 ; return
```

frame pointer register = %ebp
stack pointer register = %esp
Typical prologue/epilogue code of function using frame pointer

Rewritten Binary Code

```
.FMT: .string "%d%d%d%100$n"
foo:                      ; STACK CHANGE (0)
  pushl %ebp              ; %esp -= -4 ( -4)
  movl  %esp, %ebp        ; %ebp = %esp( -4)
  subl  $24, %esp         ; %esp -= 24 (-28)
  subl  $4, %esp          ;      -=  4 (-32)
  pushl $2                ;      -=  4 (-36)
  pushl $1                ;      -=  4 (-40)
  pushl $.FMT             ;      -=  4 (-44)
  call  safe_printf_fp    ;   += -4+4 (-44)
  addl  $16, %esp         ;     += 16 (-28)
  leave                   ;   = %ebp+4(  0)
  ret                     ;     +=  4 ( +4)
safe_printf_fp:         ;INSERTED CODES
  movl  %ebp, %eax    ;calculate
  subl  %esp, %eax    ;stack depth: %eax
  subl  $8, %eax      ; = %ebp - %esp - 8
  pushl %eax          ;call
  call  safe_printf   ;safe_printf(%eax,
  addl  $4, %esp      ;retaddr,format,...)
  ret
safe_printf:
  ...
```

Passing Stack Depth In a Function USING Frame Pointer

Original Binary Code for foo()

```
.FMT: .string "%d%d%d%100$n"
foo:
  pushl %ebp         ; setup frame pointer
  movl  %esp, %ebp   ;
  subl  $24, %esp    ; alloc local var mem
  subl  $4, %esp     ; typical pattern of
  pushl $2           ; function call
  pushl $1           ;
  pushl $.FMT        ; printf(.L0,1,2);
  call  printf       ;
  addl  $16, %esp    ;
  leave              ; reset frame pointer
  ret                ; return
```

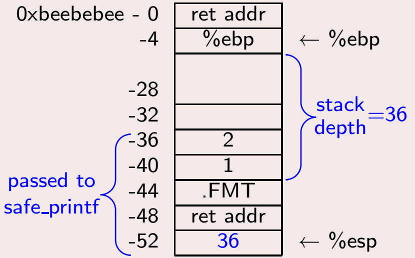
- call printf is replaced with call safe_printf_fp.
- safe_printf_fp calls safe_printf passing stack depth as an additional argument.

Rewritten Binary Code

```
.FMT: .string "%d%d%d%100$n"
foo:                      ; STACK CHANGE (0)
  pushl %ebp              ; %esp -= -4 ( -4)
  movl  %esp, %ebp        ; %ebp = %esp( -4)
  subl  $24, %esp         ; %esp -= 24 (-28)
  subl  $4, %esp          ;      -=  4 (-32)
  pushl $2                ;      -=  4 (-36)
  pushl $1                ;      -=  4 (-40)
  pushl $.FMT             ;      -=  4 (-44)
  call  safe_printf_fp    ;   += -4+4 (-44)
  addl  $16, %esp         ;     +=  16 (-28)
  leave                   ;    = %ebp+4(  0)
  ret                     ;     +=   4 ( +4)
safe_printf_fp:        ;INSERTED CODES
  movl  %ebp, %eax     ;calculate
  subl  %esp, %eax     ;stack depth: %eax
  subl  $8, %eax       ; = %ebp - %esp - 8
  pushl %eax           ;call
  call  safe_printf    ;safe_printf(%eax,
  addl  $4, %esp       ;retaddr,format,...)
  ret
safe_printf:
  ...
```

Passing Stack Depth In a Function USING Frame Pointer

Change of the Stack



- stack depth = %ebp - %esp - 8

Rewritten Binary Code

```
.FMT: .string "%d%d%d%100$n"
foo:                              ; STACK CHANGE (0)
  pushl %ebp                      ; %esp -= -4 ( -4)
  movl  %esp, %ebp                ; %ebp = %esp( -4)
  subl  $24, %esp                 ; %esp -= 24 (-28)
  subl  $4, %esp                  ;        -=  4 (-32)
  pushl $2                        ;        -=  4 (-36)
  pushl $1                        ;        -=  4 (-40)
  pushl $.FMT                     ;        -=  4 (-44)
  call  safe_printf_fp            ;     += -4+4 (-44)
  addl  $16, %esp                 ;        += 16 (-28)
  leave                           ;     = %ebp+4(  0)
  ret                             ;        +=  4 ( +4)
safe_printf_fp:                ;INSERTED CODES
  movl  %ebp, %eax              ;calculate
  subl  %esp, %eax              ;stack depth: %eax
  subl  $8, %eax                ; = %ebp - %esp - 8
  pushl %eax                    ;call
  call  safe_printf             ;safe_printf(%eax,
  addl  $4, %esp                ;retaddr,format,...)
  ret
safe_printf:
  ...
```

Passing Stack Depth In Func. NOT USING Frame Pointer

Original Binary Code

```
.FMT: .string "%d%d%d%100$n"
foo:                     ; STACK CHANGE (  0)
  subl  $12, %esp        ;        %esp = -12
  subl  $4, %esp         ;             = -16
  pushl $2               ;             = -20
  pushl $1               ; stack depth = -24
  pushl $.FMT
  call  printf
  addl  $16, %esp
  addl  $12, %esp
  ret
```

call printf is replaced with safe_printf_sp passing the corresponding stack depth value.

Rewritten Binary Code

```
.FMT: .string "%d%d%d%100$n"
foo:
  subl  $12, %esp
  subl  $4, %esp
  pushl $2
  pushl $1
  pushl $.FMT
  call  safe_printf_sp_24
  addl  $16, %esp
  addl  $12, %esp
  ret
safe_printf_sp_24:       ; INSERTED CODES
  pushl  $24             ; stack depth = 24
  call   safe_printf
  addl   $4, %esp
  ret
safe_printf:
  ...
```

Defense of indirect function calls to printf

A direct call to printf

```
addl $4, %esp
pushl $2
pushl $1
pushl $.FMT
call printf            ; printf(.FMT, 1, 2)
addl $16, %esp
```

An indirect call to printf

```
movl $printf, %eax    ; eax = printf
...
addl $4, %esp
pushl $2
pushl $1
pushl $.FMT
call *%eax            ; (*eax)(.FMT, 1, 2)
addl $16, %esp
```

- Finding indirect calls to printf by static analysis is difficult
- The analysis of parameter length of an indirect function call is same to the direct function call
- The location of a (direct or indirect) function call in static program code space is constant

Detection of Indirect Calls to printf

1. insert a copy of printf, called printf_clone into the binary program
2. replace direct calls to printf with calls to printf_clone
3. overwrite the code, jmp safe_printf_indirect at the beginning of printf function body
4. The direct printf call executes printf_clone, and
   The indirect printf call executes safe_printf_indirect

Hash Table of Parameter Length of Indirect Function Calls

- Calculate the parameter length(L) of indirect function call by static analysis on binary code.
- The location of indirect function call(IP) = the address of following machine code of the function call code, which is the return address of the function call
- Register (IP, L) into the parameter length hash table(PL).
- Insert the hash table PL into the modified binary program.

```
804838b:  83 ec 04        subl $0x4, %esp
804838e:  6a 02           pushl $0x2        --+
8048390:  6a 01           pushl $0x1          | L = 12
8048392:  68 84 84 04 08  pushl $0x8048484  --+
8048397:  8b 45 fc        movl -4(%ebp), %eax
804839a:  ff d0           call *%eax
804839c:  83 c4 10        addl $0x10,%esp   --> IP = 0x804839c
```

Calling safe_printf by safe_printf_indirect

safe_printf_indrect function

```
int safe_printf_indirect() {
  L = get_param_len(PL_HASH, return_address);
  if (L != NOT_FOUND) {
    extra_param_len = L - PRINTF_PARAM_PREFIX;  // = L - 4
    asm {
      pushl extra_param_len;
      call safe_printf;   // safe_printf(L, retaddr, fmt, ...)
    }
  } else
    asm call printf_clone;  // printf_clone(fmt, ...)
}
```

User Defined printf Function

The C Code Pattern of User Defined printf Function

```
int myprintf(int pre,char *fmt,...)
{
  va_list ap;
  va_start(fmt, ap);
  rc = vprintf(fmt, ap);
  va_end(ap);
  return rc;
}

myprintf(123, "%d%d", 1, 2);
```

- User Defined printf Function: A function F that calls vprintf with the format string and format arguments which are parameters of F.

An Example of Binary Code of myprintf

```
myprintf:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp          ; -4(%ebp) --> ap
  leal 16(%ebp), %eax    ; va_start(ap,fmt)
  movl %eax, -4(%ebp)    ; ap=&first_ext_arg
  subl $8, %esp          ;
  pushl -4(%ebp)         ; ap
  pushl 12(%ebp)         ; fmt
  call vprintf           ; vprintf(fmt, ap)
  addl $16, %esp         ;
  movl %eax, -8(%ebp)
  movl -8(%ebp), %eax
  leave
  ret
```

- ap is implemented as a pointer variable (IA32 ABI standard)
- va_end(ap) is dummy code

The Protection of User Defined printf Functions

- Kimchi detects user defined printf(udf_printf) functions by static analysis on binary code pattern,
- and registers udf_printf as a new printf function.
- Defense method of udf_printf is same to printf
  - replaces the code 'call udf_printf' with 'call safe_udf_printf'
  - inserts the binary code of safe_udf_printf into the binary program
  - udf_printf(123, "%d%d", 1, 2)
    → safe_udf_printf(8, 123, "%d%d", 1, 2)
    8 = the parameter length

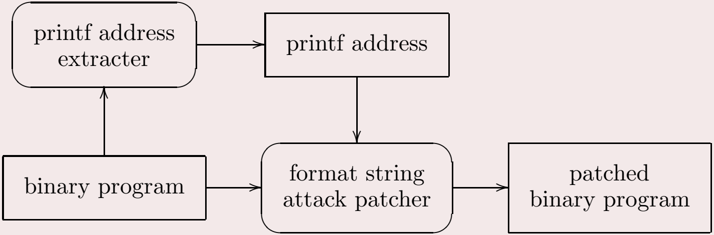
Kimchi

Kimchi Is for Machine Code's Health Improvement

Target Applications of Experimental Prototype System

IA32 ELF Executables in Linux System without Source Code Information

The Structure of `Kimchi`



Binary Rewriting Process

Format String Attack Patcher

1. the disassemble of binary codes,
2. the search of `printf` calls,
3. the construction of control flow graph(CFG),
4. the analysis of stack frame depth,
5. the construction of patch information, and
6. the creation of patched binary program.

Development Environment

- IA32 Linux System
- C Programming Language
- GNU glibc Library
- GNU binutils
  - I/O of ELF executables
- Diablo(Diablo Is A Better Link-time Optimizer)
  - disassemble of binary codes
  - static analysis of binary codes

Disassemble of Binary Codes

Kimchi implements linear sweep diassemble alogirithm.

Disassemble Alogorithms

- linear sweep disassemble algorithm
- recursive traversal disassemble algorithm
- hybrid disassemble algorithm: linear sweep + recursive traversal

Construction of Control Flow Graph

1. disassemble the binary
2. mark all basic block leaders (program entry point, successors of control transfer instructions, targets of control transfer instructions).
3. extract basic blocks (for each leader, put the instructions starting at that leader, up to but not including the next leader as a node in the CFG, the nodes are called basic blocks)
4. connect basic blocks with the right types of edges in the graph-structure

Search of printf function address

in case that libc is:

- dynamically linked
  from dynamic relocation records in ELF binary file [ELF Spec. 1995]

```
foo:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049578 R_386_GLOB_DAT    __gmon_start__
08049588 R_386_JUMP_SLOT   __libc_start_main
0804958c R_386_JUMP_SLOT   printf
```

- statically linked
  pattern matching using signature of binary codes for printf [Emmerik 1994]

```
signature of _IO_vfprintf in glibc-2.3.4/Linux i686

5589e557 565381ec bc050000 c78558fb ffff0000 0000e8XX XXXXXX8b 108b4d08
89953cfb ffff8b51 5c85d2c7 8538fbff ff000000 00750cc7 415cffff ffffbaff
ffffff42 b9ffffff ff752e8b 75088b16
```

The Rewritten Binary Program

Modification of a Binary Program

Before translation
ELF header
other sections. . .
.text section . . . call printf . . . . . . call printf . . . . . . call printf . . . . . . call printf . . .
other sections. . .

After translation
ELF header
other sections. . .
.text section . . . call safe_printf_fp . . . . . . call safe_printf_32 . . . . . . call safe_printf_64 . . . . . . call safe_printf_fp . . .
.text.safe_format section safe_printf_fp: . . . safe_printf_32: . . . safe_printf_64: . . . safe_printf: . . .
other sections. . .

- Modification of `.text` code section:
  replaces calls to `printf` with `safe_printf_*`
- Insertion of `.text.safe_format` section:
  contains `safe_printf` function bodies

The Overall Performance Overhead is Small

Test Code for Microbenchmark

```
int main(void) {
  int i;
  for (i = 0; i < 10000000; i++)
    printf("%s %s %s\n",
           "a", "b", "c");
  printf("%d\n", i);
  exit(0):
}
```

Test Environment

- Intel Pentium III 1GHz CPU, 256MB
- Single user mode in Linux/x86 with kernel-2.6.8

Performance Overhead

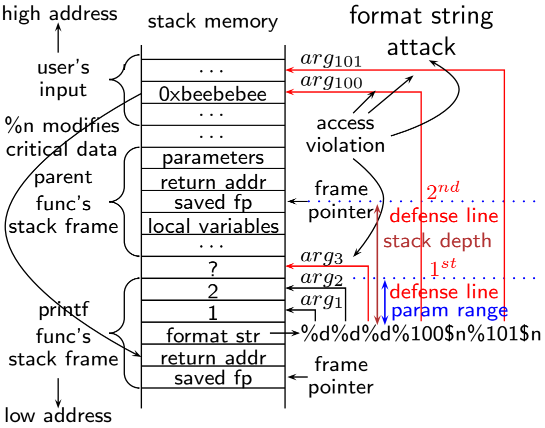
function	marginal overhead
safe_sprintf	29.5%
safe_fprintf	29.5%
safe_printf	2.2%

Just a few printf calls with non-constant format string need the defense patch in general

Program Size Overhead

Sum of code sizes of `safe_printf`, `safe_printf_fp` and a number of `safe_printf_sp_*`

Defense Idea of safe_printf with Extra Arguments



- Kimchi replaces binary codes to call printf with ones to call safe_printf
- safe_printf protects from accessing over "1st or 2nd defense line"
- stack depth as the range of parameters is passed to safe_printf when Kimchi can not determine the parameter range.
- The same defense method is applied to fprintf, sprintf, snprintf, syslog, warn, err, . . .

Kimchi

- wrapping printf() functions by binary rewriting
- parameter based protection against format string attack
- prevention of format directives' accessing parameter over its parameter range or parent's stack frame
- supports both frame pointer and stack pointer based stack frame
- supports both dynamically and statically linked binary executables
- transforms printf(buf) likes to printf("%s", buf)
- supports read-only format string
- needs to modify binary executables
- dependant to the power of static analysis of binary code pattern

libsafe

- wrapping printf() functions by dynamic linking mechanism
- parameter based protection against format string attack
- prevention of format directives' accessing parameter over parent's stack frame
- support only binary executables using stack frame pointer register

libformat

- wrapping printf() functions by dynamic linking mechanism
- format string content based protection against format string attack
- prevention of using the feature, '%n':
  violates C standard

TaintCheck

- wrapping printf() functions by runtime tracing and hooking mechanism
- traces binary code execution paths and calculates propagation of the tainted data: this slows the execution sppeed
- format string content based protection against format string attack
- prevention of using format directives propagated from external untrusted input
- prevention of using the feature, '%n'

Comparison with Previous Binary Level Defense Methods I

	Kimchi	LS	LF	TC
Dynamically linked binary support	○	○	○	○
Statically linked binary support	○	×	×	○
Frame pointer based stack frame	○	○	○	○
Stack pointer based stack frame	○	×	○	○
vprintf() family	△	○	○	○
Parameter range baed protection	○	△	×	×
Prevention of reading-memory attack	○	○	×	△
Availability of '%n' feature	○	○	×	○
Format string including external input format directives	○	○	○	×

* LS = libsafe, LF = libformat, TC = TaintCheck

Comparison with Previous Binary Level Defense Methods II

	Kimchi	LS	LF	TC
printf(buf) → printf("%s", buf)	○	×	×	×
Read-only format string support	○	×	×	○
No need of preprocessing of program	×	○	○	○
Independent to binary code pattern	×	○	○	○
Performance overhead of protection	Low	Low	Low	High

* LS = libsafe, LF = libformat, TC = TaintCheck

The proposed system tool, Kimchi

- modifies binary programs of Linux/IA32
- even if the `libc` library is statically linked to them, or
- they don't use the frame pointer register
- to defend against format string attacks in runtime.
- The performance and size overhead of modified binary program is small.

Future Work

The static analysis of

- the range of function arguments
- user defined printf functions

in the complicated binary code pattern.

References I

- U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *Procedings of the 10th USENIX Security Symposium (SECURITY-01)*, 2001.
- C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "FormatGuard: Automatic protection from printf format string vulnerabilities," in *the 10th USENIX Security Symposium*, 2001.
- George C. Necula, Scott McPeak, and Westley Weimer, CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.
- T. J. Robbins: "libformat," http://box3n.gumbynet.org/~fyre/software/.

References II

N. Singh and T. Tsai, "Libsafe 2.0: Detection of format string vulnerability exploits," July 27 2001.

Newsome, J., Song, D., Dynamic taint analysis for automatic detection, analysis, and signature gerneration of exploits on commodity software. In: Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05). (2005)

M. Prasad and T. cker Chiueh, "A binary rewriting defense against stack-based buffer overflow attacks," in *the Proceedings of USENIX 2003 Annual Technical Conference*, 2003.

G. A. Kildall, "A unified approach to global program optimization," *In ACM Symposium on Principles of Programming Languages*, pp. 194–206, oct 1973.

References III

- T. I. S. T. Committee, “Executable and linking format (ELF) specification, version 1.2,” 1995.
- M. V. Emmerik, “Signatures for library functions in executable files,” Tech. Rep. FIT-TR-1994-02, 14, 1994.