Source Code Analysis Tools - Example Programs

Cigital, Inc.

2006-07-06 L2 / L[1]

These example programs demonstrate flaws that may (or may not) be detected by security scanners for C/C++ software. The examples are small, simple C/C++ programs, each of which is meant to evaluate some specific aspect of a security scanner's performance. Overall, the evaluation programs can be categorized as programs used to evaluate the detection of potential vulnerabilities and those used to evaluate resilience against false alarms.

Example 1

```
/**

This file is meant to test whether a scanner can perform pointer
alias analysis. Since that capability is generally only useful if the
scanner provides some dataflow analysis capabilities, dataflow
analysis is needed too.

The variable that determines the size of a string copy is untainted,
but alias analysis is needed to determine this.

*/

int main(int argc, char **argv)
{
  int len = atoi(argv[1]);
  int *lenptr_1 = &len;
  int *lenptr_2 = lenptr_1;
  char buffer[24];

  *lenptr_2 = 23;
  strncpy(buffer, argv[2], *lenptr_1);
}
```

Example 2

```
/* unexploitable strcpy #1 */

/* This program contains a buffer overflow, but the overflowing data
   isn't controlled by the attacker. Ideally, a scanner should either not
   report a buffer overflow associated with this strcpy, or at most report
   a problem with lower severity than a strcpy whose argument is attacker-
   controlled.
*/


main()
{
  char *buffer = (char *)malloc(10 * sizeof(char));

  strcpy(buffer, "foooooooooooooooooooooooooooooooooooooooooooooooooo");
}
```

Example 3

```
/* unexploitable strcpy #2 */
/* This program contains a buffer overflow, but the overflowing data
   isn't controlled by the attacker. Ideally, a scanner should either not
```

```
   report a buffer overflow associated with this strcpy, or at most report
   a problem with lower severity than a strcpy whose argument is attacker-
   controlled.

   The program is similar to const_str1.c, but it presents a slightly
   harder problem for the scanner. In const_str1.c, and scanner could
   notice that the argument to strcpy is a constant string by looking for the
   quote symbol that follows the open parenthesis after the name of the
   function. In this program, some sort of dataflow analysis is needed
   (taint checking should be enough).
*/


void func(char *foo)
{
  char *buffer = (char *)malloc(10 * sizeof(char));

  strcpy(buffer, foo);
}

main()
{
  func ("fooooooooooooooooooooooooooooooooooooooooooooooooo");
}
```

Example 4

```
/* unexploitable strcpy #3 */

/*
  This is another buffer overflow using a non-user-defined. Here, the
  constant string is placed into a variable rather than being passed as
  a function argument like in const_str2.c. However, taint analysis should
  still be enough to let the scanner recognize that the overflowing string
  is not user-controlled.

  A scanner should either not report a buffer overflow associated with
  this strcpy, or report a problem with lower severity than a strcpy whose
  argument is attacker-controlled.
*/

main()
{
  char *foo = "fooooooooooooooooooooooooooooooooooooooooooooooooooo";
  char *buffer = (char *)malloc(10 * sizeof(char));
  strcpy(buffer, foo);
}
```

Example 5

```
/* believed unexploitable open/write */

/* This program ensures that stdin, stdout and stderr are accounted for,
   and then opens a file, ensuring that access checks are performed on
   the actual object being opened. The program doesn't set the umask, but that
   isn't necessary because the umask only affects the permissions of newly
   created files, and in this program open is called without the O_CREAT
   flag and therefore will only open a pre-existing file.

   A scanner should not report TOCTOU vulnerabilities, file descriptor
   vulnerabilities or umask-related vulnerabilities.
*/

/* ex_02.c */
#include
#include
```

```
#include
#include
#include
#include

int
main (int argc, char * argv [])
{
    struct stat st;
    int fd;
    FILE * fp;


    while((fd = open("/dev/null", O_RDWR)) == 0 || fd == 1 || fd == 2) ;
    if (fd > 2)
      close(fd);

    if (argc != 3) {
        fprintf (stderr, "usage : %s file message\n", argv [0]);
        exit(EXIT_FAILURE);
    }
    if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
        fprintf (stderr, "Can't open %s\n", argv [1]);
        exit(EXIT_FAILURE);
    }
    fstat (fd, & st);
    if (st . st_uid != getuid ()) {
        fprintf (stderr, "%s not owner !\n", argv [1]);
        exit(EXIT_FAILURE);
    }
    if (! S_ISREG (st . st_mode)) {
        fprintf (stderr, "%s not a normal file\n", argv[1]);
        exit(EXIT_FAILURE);
    }
    if ((fp = fdopen (fd, "w")) == NULL) {
        fprintf (stderr, "Can't open\n");
        exit(EXIT_FAILURE);
    }
    fprintf (fp, "%s", argv [2]);
    fclose (fp);
    fprintf (stderr, "Write Ok\n");
    exit(EXIT_SUCCESS);
}
```

Example 6

```
/* variable-sized buffer that syntactically looks like fixed-sized buffer #1 */

/* Many security scanners generate a warning when they see a fixed-sized
   buffer. This test program declares a variable-sized buffer based on
   the length of the string that's going to be copied into it, but it
   uses a syntax more commonly associated with fixed-sized buffers. It's
   meant to determine whether a scanner detects fixed-sized buffers by looking
   for square brackets after the variable name or whether it actually parses
   the declaration.

   A scanner should not complain about a fixed-sized buffer being used
   in this program.
*/

#include

void func(char *src)
{
  char dst[(strlen(src) + 1) * sizeof(char)];
  strncpy(dst, src, strlen(src) + sizeof(char));
  dst[strlen(dst)] = 0;
```

```
}
```

Example 7

```
/* variable-sized buffer that syntactically looks like fixed-sized buffer #2 */

/* This is another variant of a variable-sized buffer being made to
   syntactically resemble a fixed-sized buffer. It has the added twist
   the buffer might be to small if useString is called incorrectly, in spite
   of which there is no buffer overflow here because useString -is- called
   correctly (and is inaccesible from other source files).

   A scanner should not complain about a fixed-sized buffer or a potential
   buffer overflow.
*/

#include

static void useString(size_t len, char *src)
{
  char dst[(len+1) * sizeof(char)];
  strncpy(dst, src, strlen(src));
  dst[strlen(src)] = 0;
}

void func(char *src)
{
  size_t len = strlen(src);

  useString(len, src);
}
```

Example 8

```
/* This program opens a file with a fixed name in a directory that
   shouldn't normally be accessible to an attacker. If, for some reason,
   the attacker has gained write access to /etc, this program could be used
   to overwrite files in other places, but the vulnerability is less
   serious than it would be if it opened a file in a directory that's normally
   writable.
*/

#include
#include
#include
#include


main()
{
  int fd;
  FILE *fp;

  /* no file descriptor confusion */

  while((fd = open("/dev/null", O_RDWR)) == 0 || fd == 1 || fd == 2) ;
  if (fd > 2)
    close(fd);


  /* set umask */

  umask(022);

  /* file is in user-unwritable directory */

  fp = fopen("/etc/importantFile", "w");
```

```
  fclose(fp);
}
```

Example 9

```
typedef char gchar;

void func()
{
  gchar buf[10];
}
```

Example 10

```
#include

/** This program doesn't contain an integer overflow on line 15
    because the length of the variable len is checked. It's meant
    to complement overflow.c, to check if buffer overflow warnings
    for that program are just vacuously triggered by the read()
    call or if the scanner is actually spotting the overflow.

    A scanner shouldn't complain about an integer overflow on line
    15 or a buffer overflow on line 16.
*/

void func(int fd)
{

 char *buf;
 size_t len;

 read(fd, &len, sizeof(len));

 /* check the maximum length. No need to check for negative numbers since
    size_t is unsigned already. */

 if (len > 1024)
   return;

 buf = malloc(len+1);                    // line 15
 read(fd, buf, len);                     // line 16
 buf[len] = '\0';
}
```

Example 11

```
/* This program complements truncated.c, which is taken from the linux
   secure programming HOWTO. It avoids the integer truncation problem of
   truncated.c, and it's meant to test whether a scanner that reports a
   buffer overflow for truncated.c is doing so vacuously or whether it
   actually noticed the possible integer truncation.

   In this file, we read a tainted integer and use it to determine the size
   of a subsequent read of a tainted string. But the buffer recieving the
   data during the second read is allocated according to user provided length,
   and read will only put that many bytes in the buffer, so there should
   be no overflow.

   In this particular variant of the program, the user has defined his
   own version of the malloc function which takes an int argument and
   thereby creates the possibility of an integer truncation vulnerability.
   However, the program casts "len" to an integer and thereby ensures that the
   second argument to read (line 18) is the same number as the argument of
```

```
   mymalloc on line 17.

   Ideally, a security scanner should not report a possible bounds
   violation on line 17 or a buffer overflow on line 18.
*/


#include
#include

void *mymalloc(unsigned int size) { return malloc(size); }

void func(int fd)
{

  char *buf;
  size_t len;
  int actual_len;

  read(fd, &len, sizeof(len));


  actual_len = len;

  buf = mymalloc(actual_len);                    // line 17
  read(fd, buf, actual_len);                     // line 18
}
```

Example 12

```
/* This program complements truncated.c, which is taken from the linux
   secure programming HOWTO. It avoids the integer truncation problem of
   truncated.c, and it's meant to test whether a scanner that reports a
   buffer overflow for truncated.c is doing so vacuously or whether it
   actually noticed the possible integer truncation.

   In this program, the developer has defined a custom
   version of the malloc function which takes an int argument, and
   thereby creates the possibility of an integer truncation vulnerability,
   but bounds-checking prevents the malloc on line 1 from seeing
   a different length value than the read on line 16.

   This program differs from nottruncated2.c because both
   mymalloc and read take the original user-controlled size_t len as an
   argument, but those calls are unreachable for values of len that would
   cause truncation problems.

   Ideally, a security scanner should not report a possible bounds
   violation on line 15 or a buffer overflow on line 16.
*/


#include
#include

void *mymalloc(unsigned int size) { return malloc(size); } // line 1

void func(int fd)
{

  char *buf;
  size_t len;

  read(fd, &len, sizeof(len));


  if (len > MAXINT)
    return;
```

```
  buf = mymalloc(len);                            // line 15
  read(fd, buf, len);                             // line 16
}
```

Example 13

```
/*
   This program avoids a sign error by checking of the variable len is
   negative. It complementes signedness_1.c, where an attacker can create
   a buffer overflow by specifying a negative number for len.

   A scanner should not report a buffer overflow on line 11.
*/

void func(int fd)
{
 char *buf;
 int i, len;

 read(fd, &len, sizeof(len));

 if (len < 0 || len > 7999) { error("too large length"); return; }

 buf = malloc(8000);
 read(fd, buf, len);                // line 11
}
```

Example 14

```
/*
  This program uses strncpy and strncat safely, without introducing a
  buffer overflow. It's meant to check whether a scanner warns vacuously
  about strncpy and strncat, or if it actually checks whether the sizes are
  OK and whether the buffer is terminated after the strncpy.

  A scanner should not report a buffer overflow on line 5 or line 7.
*/


main(int argc, char **argv)
{
  char *buffer = (char *)malloc(25);

  strncpy(buffer, argv[1], 10);         // line 5
  buffer[10] = 0;
  strncat(buffer, argv[2], 10);         // line 7
}
```

Example 15

```
/*
   This use of strcpy ensures that the buffer is large enough to
   accomodate the string being copied. The dataflow analysis needed to
   determine whether the strcpy is safe is somewhat more complex that
   in strsave.c

   A scanner should not warn of a buffer overflow error on line 5.
*/


#include

static void copyString(char *dst, char *src)
{
```

```
  strcpy(dst, src);                           // line 5
}

char *strsave(char *src)
{
  size_t len = strlen(src);
  char *result = (char *)malloc((len + 1) * sizeof(char));

  if (result)
    copyString(result, src);

  return result;
}
```

Example 16

```
/* believed safe invocation of strcpy */

/* This use of strcpy ensures that the buffer is large enough to
   accomodate the string being copied. The dataflow analysis needed
   to verify this mey be too complex to be accomplished with simple
   taint checking.

   A scanner should not warn of a buffer overflow error on line 9
*/

#include

char *strsave(char *src)
{
  size_t len = strlen(src);
  char *result = (char *)malloc((len+1) * sizeof(char));

  if (result)
    strcpy(result, src);                            // line 9

  return result;
}
```

Example 17

```
/**
   In this program, the target string is properly terminated but
   the terminating null is added before the strncpy(), which might
   fool a scanner into thinking that the buffer is unterminated.

   A scanner should not complain about an unterminated strcpy().
*/

void func(char *str)
{
  char target[(strlen(str) + 1) * sizeof(char)];
  target[strlen(str)] = 0;
  strncpy(target, str, strlen(str));
}
```

Example 18

```
/**
   The catch block in this program contains an unexploitable format-string
   vulnerability. The idea of this test to see whether the scanner can track
   taint through the exception-handling mechanism. Ideally, the warning given
   by the scanner for line 31 should have lower severity than the
   corresponding (exploitable) format-string vulnerability in except.c
*/
```

```
#include
#include
#include

void func()
{
  char buffer[1024];
  printf("Please enter your user id :");
  fgets(buffer, 1024, stdin);

  if (!isalpha(buffer[0]))
  {
    char errormsg[1044];

    strcpy(errormsg, "that isn't a valid ID");
    throw errormsg;
  }

}


main()
{
  try
  {
    func();
  }
  catch(char * message)
  {
    fprintf(stderr, message);                // line 31
  }
}
```

Example 19

```
char *stringcopy(char *str1, char *str2)
{
  while (*str2)
    *str1++ = *str2++;

  return str2;
}

main(int argc, char **argv)
{
  char *buffer = (char *)malloc(16 * sizeof(char));
  stringcopy(buffer, argv[1]);
  printf("%s\n", buffer);
}
```

Example 20

```
/* didn't check for file descriptor tricks */
/* If this is a setuid program, the attacker can exec() it after closing
   file descriptor 2. The next time the program opens a file, the file
   is associated with file descriptor 2, which is stderr. All output
   directed to stderr will go to the newly opened file. In this example, the
   attacker creates a symbolic link to the file that is to be overwritten.
   The name of the link contains the data to be written. When the
   program detects the symbolic link, it prints an error message and exits
   (line 32), but the error message, which contains the symbolic-link name
   supplied by the attacker, is written into the targeted file.
*/
```

```
/* ex_02.c */

#include
#include
#include
#include
#include
#include

int
main (int argc, char * argv [])
{
    struct stat st;
    int fd;
    FILE * fp;

    if (argc != 3) {
        fprintf (stderr, "usage : %s file message\n", argv [0]);
        exit(EXIT_FAILURE);
    }
    if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
        fprintf (stderr, "Can't open %s\n", argv [1]);
        exit(EXIT_FAILURE);
    }
    fstat (fd, & st);
    if (st . st_uid != getuid ()) {
        fprintf (stderr, "%s not owner !\n", argv [1]);
        exit(EXIT_FAILURE);
    }
    if (! S_ISREG (st . st_mode)) {
      fprintf (stderr, "%s not a normal file\n", argv[1]); // line 32
        exit(EXIT_FAILURE);
    }
    if ((fp = fdopen (fd, "w")) == NULL) {
        fprintf (stderr, "Can't open\n");
        exit(EXIT_FAILURE);
    }
    fprintf (fp, "%s", argv [2]);
    fclose (fp);
    fprintf (stderr, "Write Ok\n");
    exit(EXIT_SUCCESS);
}
```

Example 21

```
/* stat called on filename */
/* This is a simple race condition, allowing the attacker to change the file
   named in argv[1] to a symbolic link after it's tested but before the file
   is opened.

   Many scanners detect the call to stat() on line 23, and while stat() is
   almost certainly a sign of trouble in this particular context, it
   needn't always be. A better scanner would actually detect the race
   condition between the open on line line 14 and the stat on line 23.
*/

#include
#include
#include
#include
#include
#include

int
main (int argc, char * argv [])
{
    struct stat st;
```

```
    int fd;
    FILE * fp;

    while((fd = open("/dev/null", O_RDWR)) == 0 || fd == 1 || fd == 2); //ln 14
    if (fd > 2)
      close(fd);

    if (argc != 3) {
        fprintf (stderr, "usage : %s file message\n", argv [0]);
        exit(EXIT_FAILURE);
    }
    stat (argv[1], & st);                                       // line 23
    if (st . st_uid != getuid ()) {
        fprintf (stderr, "%s not owner !\n", argv [1]);
        exit(EXIT_FAILURE);
    }
    if (! S_ISREG (st . st_mode)) {
        fprintf (stderr, "%s not a normal file\n", argv[1]);
        exit(EXIT_FAILURE);
    }
    if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
        fprintf (stderr, "Can't open %s\n", argv [1]);
        exit(EXIT_FAILURE);
    }
    if ((fp = fdopen (fd, "w")) == NULL) {
        fprintf (stderr, "Can't open\n");
        exit(EXIT_FAILURE);
    }
    fprintf (fp, "%s", argv [2]);
    fclose (fp);
    fprintf (stderr, "Write Ok\n");
    exit(EXIT_SUCCESS);
}
```

Example 22

```
/**
   The catch block in this program contains an exploitable format-string
   vulnerability. The idea of this test to see whether the scanner can track
   taint through the exception-handler. Ideally, the scanner should report
   a format string vulnerability on line 32, but not report the unexploitable
   format string vulnerability in the complementary program unexcept.c.
*/



#include
#include
#include

void func()
{
  char buffer[1024];
  printf("Please enter your user id :");
  fgets(buffer, 1024, stdin);

  if (!isalpha(buffer[0]))
  {
    char errormsg[1044];

    strncpy(errormsg, buffer, 1024);         // guaranteed to be terminated
    strcat(errormsg, " is not a valid ID");  // we have room for this
    throw errormsg;
  }

}
```

```
main()
{
  try
  {
    func();
  }
  catch(char * message)
  {
    fprintf(stderr, message);                // line 32
  }
}
```

Example 23

```
/* If this is a setuid program, the attacker can exec() it after closing
   file descriptor 2. The next time the program opens a file, the file
   is associated with file descriptor 2, which is stderr. All output
   directed to stderr will go to the newly opened file. In this example, the
   attacker creates a symbolic link to the file that is to be overwritten.
   The name of the link contains the data to be written. When the
   program detects the symbolic link, it prints an error message and exits,
   but the error message, which contains the symbolic link name supplied by
   the attacker, is written into the targeted file. This isn't much different
   from ex_02.c, but the latter program was found on the web claiming to
   be a secure way of opening files. This program is somewhat simpler
   and, for some scanners, might make it easier to tell what the scanner
   is printing warnings about.
*/



#include
#define DATAFILE "/etc/aDataFile.data"


main(int argc, char **argv)
{
  FILE *sensitiveData = NULL;
  FILE *logFile = NULL;

  /* Forgot to account for files 0-2, could be opening stderr. */

  sensitiveData = fopen(DATAFILE, "w");

  if (!sensitiveData)
  {
    fprintf(stderr, "%s: failed to open %s\n",
     argv[0], DATAFILE);
    exit(1);
  }

  logFile = fopen(argv[1], "w");

  if (!logFile)
  {
    fprintf(stderr, "%s: failed to open %s\n",
     argv[0], argv[1]);
    exit(1);
  }
}
```

Example 24

```
/*

buffer overflow using a custom version of the strcpy() function.
```

```
*/


char *stringcopy(char *str1, char *str2)
{
  while (*str2)
    *str1++ = *str2++;

  return str2;
}

main(int argc, char **argv)
{
  char *buffer = (char *)malloc(16 * sizeof(char));
  stringcopy(buffer, argv[1]);
  printf("%s\n", buffer);
}
```

Example 25

```
/* This program tests the scanner's ability to handle preprocessor
   directives.
*/

#include

#define SAFESTRCPY(a,b,c) strncpy(a, b, c)
#define FASTSTRCPY(a,b,c) strcpy(a, b)

main(int argc, char **argv)
{
  size_t size = strlen(argv[3]);
  char *buffer = (char *)malloc(1024);

#ifdef PARANOID
  SAFESTRCPY(buffer, argv[3], size+sizeof(char));
#else
  FASTSTRCPY(buffer, argv[3], size+sizeof(char));
#endif
}
```

Example 26

```
/* Secure-Programs-HOWTO/dangers-c.html */

/* In this program, an attacker can supply a large value of len which
   overflows to zero on line 14. Since the subsequent read on line 15
   uses the original value of len, the read can overflow the buffer.

   Many scanners will flag the read no matter what, which is useful but
   doesn't reflect what this program is trying to test. The complementary
   program notoverflow.c is meant to check whether a scanner is actually
   detecting the possible overflow.
*/

#include

void func(int fd)
{

/* 3) integer overflow */
 char *buf;
 size_t len;

 read(fd, &len, sizeof(len));

 /* we forgot to check the maximum length */
```

```
 buf = malloc(len+1);                             // line 14
 read(fd, buf, len);                              // line 15
 buf[len] = '\0';
}
```

Example 27

```
/* from Secure-Programs-HOWTO/dangers-c.html */
/* In this example, the attacker-controlled number "len" is read as an integer,
   and even though there is a test to check if it's greater than
   the length of the buffer, a negative value for len will be converted to
   a large positive value when it gets cast to an unsigned integer in the
   second call to read.
*/


void func(int fd)
{
/* 1) signedness - DO NOT DO THIS. */
 char *buf;
 int i, len;

 read(fd, &len, sizeof(len));

 /* OOPS!  We forgot to check for < 0 */
 if (len > 8000) { error("too large length"); return; }

 buf = malloc(len);
 read(fd, buf, len); /* len casted to unsigned and overflows */
}
```

Example 28

```
/* This is a simple resource-spoofing vulnerability where the characteristics
   of a fopened file are completely unchecked. (Often this would be called a
   race condition as well, but technically it isn't since the necessary checks
   are missing entirely.) First-generation scanners would be expected to
   generate warnings on this file because of the fopen(). This test is meant
   for scanners that don't warn about anything un ex2_unex.c; it checks whether
   they just ignore open() calls altogether (ignoring open() isn't what
   ex2_unex is testing for, needless to say).
*/

#include

void func()
{
  FILE *aFile = fopen("/tmp/tmpfile", "w");
  fprintf(aFile, "%s", "hello world");
  fclose(aFile);
}
```

Example 29

```
/* does the scanner understand preprocessor directives? */

/* This file tries to fool the scanner by making "strcpy" look like a variable
   instead of a function.
*/

#define STRINGCOPY strcpy

int main(int argc, char **argv)
{
```

```
  char *buffer = (char *)malloc(1024);
  STRINGCOPY(buffer, argv[3]);
}

void func()
{
  /* ideally this should not generate a warning because "strcpy" is
     just being used as the name of a variable (and in fact it's dead
     code).
  */

  int strcpy = 0;
  strcpy = strcpy + 1;
}
```

Example 30

```
/*
  In this program srncat is called ten times in a loop, but the buffer
  recieving that data isn't big enough, so there's a potential buffer
  overflow on line 9.
*/


main(int argc, char **argv)
{
  char *buffer = (char *)malloc(11);
  int i;

  buffer[0] = 0;

  for (i = 0; i < 10; i++)
    strncat(buffer, argv[i], 10);    // line 9
}
```

Example 31

```
/* Technically the buffer in this program has enough room for all the
   strncats, but the programmer forgot to terminate the buffer before
   the strncats start. Therefore line 7 contains a potential buffer
   overflow.
*/


main(int argc, char **argv)
{
  char *buffer = (char *)malloc(101);
  int i;

  for (i = 0; i < 10; i++)
    strncat(buffer, argv[i], 10);    // line 7
}
```

Example 32

```
/* aonther strncat to into an unterminated buffer. */

main(int argc, char **argv)
{
  char *buffer = (char *)malloc(101);

  strncat(buffer, argv[2], 90);
}
```

Example 33

```
/* forgot to null-terminate the strncpy */
/* strncpy doesn't automatically null-terminate the string being copied
   into. In this example, the attacker supplies an argv[1] of length ten
   or more. In the subsequent strncat, data is copied not to buffer[10]
   as the code suggests, but to the first location to the left of buffer[0]
   that happens to contain a zero byte.
*/

main(int argc, char **argv)
{
  char *buffer = (char *)malloc(101);

  strncpy(buffer, argv[1], 10);
  strncat(buffer, argv[2], 90);
}
```

Example 34

```
/*
   In this example, the attacker controls the third argument of strncpy,
   making it unsafe.
*/


#include

main(int argc, char **argv)
{
  int incorrectSize = atoi(argv[1]);
  int correctSize = atoi(argv[2]);
  char *buffer = (char *)malloc(correctSize+1);


  /* number of characters copied is based on user-supplied value */

  strncpy(buffer, argv[3], incorrectSize);
}
```

Example 35

```
/* Secure-Programs-HOWTO/dangers-c.html */

/* This program contains an integer truncation error. Superficially it looks
   like a safe program even though the variable len is tainted and
   len is used to determine the number of bytes read on line 18. It seems
   as though the buffer is large enough to accomodate whatever data ends
   up being placed there by the read statment. However, the program has
   a customized malloc function that takes an int argument, so in reality
   the malloc on line 3 doesn't always see the same argument as the read on
   line 18. A value of len larger than 2*MAXINT allows a buffer overflow on
   line 18.

   This example is somewhat contrived because of the large amount of memory
   that would have to be allocated for an exploit to succeed. On many
   architectures, len cannot be greater than 2*MAXINT.
*/

#include

void *mymalloc(unsigned int size) { return malloc(size); }   // line 3

void func(int fd)
{

/* An example of an ERROR for some 64-bit architectures,
    if "unsigned int" is 32 bits and "size_t" is 64 bits: */
```

```
  char *buf;
  size_t len;

  read(fd, &len, sizeof(len));

  /* we forgot to check the maximum length */

  /* 64-bit size_t gets truncated to 32-bit unsigned int */
  buf = mymalloc(len);
  read(fd, buf, len);                                         // line 18
}
```

Example 36

```
/* based on the incorrect statement: "umask sets the umask to mask & 0777."
   in the umask man page.
*/

/* In reality umask sets the mask to 0777 & ~mask, which is also
   contrary to the convention for chmod that most people are accustomed to.
   (However, the correct usage is given lower down on the umask man page).
   Below, the programmer uses umask to give the rest of the world full access
   to the newly created file while denying access to him or herself, which
   can safely be assumed to be a programming error.


   Difficulty level: 1
*/



#include
#include
#include
#include

main()
{

  int fd;
  FILE *fp;

  /* no file descriptor confusion */

  while((fd = open("/dev/null", O_RDWR)) == 0 || fd == 1 || fd == 2) ;
  if (fd > 2)
    close(fd);


  umask(700);  /* set permissions to ----rwxrwx */

  /* file is in user-unwritable directory */

  fp = fopen("/etc/importantFile", "w");

  fclose(fp);
}
```

Example 37

```
/* forgot to set umask */
/* umask() controls the permissions of created by the open call, but the
   permission mask is passed to the child process in an exec(). If this
   is a setuid program, the attacker can set a permission mask that makes
   these files world-writable, but the new file may be a system-critical
   one. In this program, the programmer uses the umask that existed when
```

```
   the program was exec()ed, but that umask might be controlled by an
   attacker.
*/

#include
#include
#include
#include


main()
{

  int fd;
  FILE *fp;

  /* no file descriptor confusion */

  while((fd = open("/dev/null", O_RDWR)) == 0 || fd == 1 || fd == 2) ;
  if (fd > 2)
    close(fd);


  /* file is in user-unwritable directory */

  fp = fopen("/etc/importantFile", "w");

  fclose(fp);
}
```

Example 38

```
/* Here, the developer is getting a pathname as an argument and wants
   to find the first path component. The error is that the path
   in str might start with a '/', in which case len is zero and
   len-1 is the largest value possible for a size_t. In that particular
   case the strncpy in the else clause is no safer than a strcpy.
*/


#include

void func(char *str)
{
  char buf[1024];

  size_t len;
  char *firstslash = strchr(str, '/');

  if (!firstslash)
    strncpy(buf, str, 1023);  /* leave room for the zero */
  else
  {
    len = str - firstslash;     /* length of the first path component */

    if (len > 1023)
      len = 1023;

    strncpy(buf, str, len-1);   /* cut the slash off. Only copy len-1
       characters to avoid zero padding.  */
    buf[len] = 0;
  }
}
```

Example 39

```
/* The principle here is that incorrectly casting a pointer to a C++
```

```
   object potentially breaks the abstraction represented by that object,
   since the (non-virtual) methods called on that object are determined
   at compile-time, while the actual type of the object might not be
   known until runtime. In this example, a seemingly safe strncpy causes
   a buffer overflow. (In gcc the buffer overflows into object itself
   and then onto the stack, for this particular program. With some compilers
   the overflow might modify the object's virtual table.)

   It's hard to say what a scanner should flag in this test file. In my
   opinion the only casts allowed should be virtual member functions that
   cast the this pointer to the class that owns them (e.g., As()
   functions) and I think that prevents this type of vulnerability.
*/



#include
#include


class Stringg
{

};

class LongString: public Stringg
{
private:

  static const int maxLength = 1023;
  char contents[1024];

public:

  void AddString(char *str)
  {
    strncpy(contents, str, maxLength);
    contents[strlen(contents)] = 0;
  }
};


class ShortString: public Stringg
{
private:

  static const int maxLength = 5;
  char contents[6];

public:

  void AddString(char *str)
  {
    strncpy(contents, str, maxLength);
    contents[strlen(contents)] = 0;
  }
};

void func(Stringg *str)
{
  LongString *lstr = (LongString *)str;
  lstr->AddString("hello world");
}

main(int argc, char **argv)
{
  ShortString str;

  func(&str);
```

```
}
```

Cigital, Inc. Copyright

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

1. mailto:copyright@cigital.com