| galois |

Lab: Type Safety

NFS forbids users from mounting a disk remotely with root privileges. Eventually, attackers figured out that they could specify a UID of 65536, which would pass the security checks that prevent root access. This UID would get assigned to an unsigned short integer and be truncated to a value of 0. Therefore, attackers could assume root's identity of UID 0 and bypass the protection. The flawed code that allowed this is paraphrased by the following, written in file `nfs.c`:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdint.h>

/* high number passed to uid truncates - choose right one to get low number */
void assume_privs (uint16_t uid) {
   printf("uid %u is not root so is granted mount privilege\n", uid);
   seteuid(uid);
   setuid(uid);
}

void become_user (uint32_t uid) {
   if (uid == 0) {
      printf("root is not allowed\n");
      exit(0);
   } else {
      printf("User %d is OK\n",uid);
   }
   assume_privs(uid);
}

int main (int argc, char **argv) {
   become_user(atoi(argv[1]));
}
```

The problem is that 32 bit `uid` in `become_user` gets truncated to 16 bits when passed to `assume_privs`. Compile this and run it like this:

```
[prompt]$ nfs 0
root is not allowed
[prompt]$ nfs 1
User 1 is OK
uid 1 is not root so is granted mount privilege
[prompt]$ nfs 65536
User 65536 is OK
uid 0 is not root so is granted mount privilege
```

But `uid` 0 is root. Of course, Cryptol does not support writing such code due to its very strong typing. However it can simulate the above C code by passing uid % 65536 as in the following, written in file `nfs.cry`:

```
assume_privs : [32] -> [32]
assume_privs uid =
   if uid == 0 then 1 else 2

become_user : [32] -> [32]
become_user uid =
   if uid == 0 then 3
   else assume_privs (uid % 65536)

request_privs : [32] -> [32]
request_privs uid = become_user uid
```

Where, for convenience, `assume_privs` returns 1 if root is allowed to assume privileges, and 2 if a supposed non-root user is allowed. Also, `become_user` returns 3 if the input `uid` is root's, otherwise returns `assume_privs`' output. Run Cryptol, load `nfs.cry` and execute the following:

```
Main> request_privs 0
0x00000003
Main> request_privs 1
0x00000002
Main> request_privs 65536
0x00000001
Main>
```

Exercise 1:

In `nfs.cry` add a property named `prf` with `uid` as input that will expose inputs to `request_privs` that are numbers greater than 0 yet cause `assume_privs` to grant permission. For example, run it like this:

```
Main> :s satNum=9
Main> :sat prf
Satisfiable
prf 4294901760 = True
prf 65536 = True
prf 196608 = True
prf 458752 = True
prf 983040 = True
prf 2031616 = True
prf 4128768 = True
prf 8323072 = True
prf 16711680 = True
Models found: 9
(Total Elapsed Time: 0.014s, using "Z3")
```

All of which are multiples of 65536. ■

Next show that the C code is unsafe via a saw script. The C code must be rewritten slightly to be compatible with nfs.cry and nfs.cry must be slightly rewritten as well. The revised C code, in file `nfs_1.c`, is this:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdint.h>

/* high number passed to uid truncates - choose right one to get low number */
uint32_t assume_privs (uint16_t uid) {
   if (uid == 0) return 1; else return 2;
}
```

```
uint32_t become_user (uint32_t uid) {
   if (uid == 0) return 3;
   else return assume_privs(uid);
}
int main (int argc, char **argv) {
   become_user(atoi(argv[1]));
}
```

The revised Cryptol code replaces `uid % 65536` with `uid` and, therefore, is correct. All functions are also safe.

Exercise 2:

Run `clang` to get the llvm bitcode file, call it `nfs.bc`, for the revised C code. Create a SAW file, `nfs.saw`, that either proves `become_user` in C is equivalent to `become_user` in Cryptol or finds a counterexample. Show the result of `saw nfs.saw`. ■

Situations in which a function takes a `uint64_t` length parameter, but is passed a signed integer, such as `char`, that can be influenced by users, can be disastrous. In C, builtins which have to be used carefully in this regard are `read()`, `recvfrom()`, `memcpy()`, `memset()`, `bcopy()`, `snprintf()`, `strncat()`, `strncpy()`, and `malloc()`. If users can coerce a program into passing in a negative value, a function interprets it as a large positive value, which could lead to an exploitable condition.

As an example, consider the C code below, which is in file `type2.c`. That code is intended to read data from a file into a buffer space of up to 32 bytes to a user who requests it using function `read_user_data`. The input to `read_user_data` is intended to be a positive number but suppose the input is -1. The request is passed to `get_user_length`, which is simplified from what it was in the wild, and just returns the requested value. The value `0xFFF…` is returned to `length`, which is of type `char`, a signed type. But since `length` is a signed variable, the following check whether `length` is greater than 32 fails allowing the program to continue. Hence, a read request is made using `local_read`, a stripped down version of what appeared in the wild. But, since parameter `count` is unsigned, the negative number passed to it becomes a large positive number. The result is a read of a large number of bytes to the 32 byte buffer causing a potentially disastrous leakage of information.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

/* Say count is passed -1.  Conversion to uint64_t goes from something like
   0xFF to 0xFFFFFFFFFFFFFFFF but then gets interpreted as a large positive
   number. The fully implemented function would then read a large number of
   bytes into the buffer buf */
uint64_t local_read(uint16_t fd, uint8_t *buf, uint64_t count) {
  printf("will read %lu bytes\n", count);  fflush(stdout);
  return 0;
}

/* OK if x -1 – this looks strange but paraphrases relevant part  of the
   code from the wild */
uint8_t get_user_length(uint64_t x) {  return (uint8_t)x;  }
```

```
/* attempt to read 32 bytes into a buffer - input is the number of bytes
   if the # bytes is too large for the buffer no data is read - otherwise
   local_read is called to fill the buffer - fd is imagined to be a file
   descriptor */
uint64_t read_user_data(uint16_t fd) {
   char length;                          /* signed - -1 is 0xFF */
   uint8_t buffer[32];

   length = get_user_length(fd);
   printf("length=%d\n", length); fflush(stdout);

   if (length > 32) {
      printf("not enough room in buffer\n"); fflush(stdout); return -1; }

   if (local_read(fd, buffer, length) < 0) { perror("read: %m"); return -1; }
   printf("success\n");  fflush(stdout);
   return 0;
}

int main (int argc, char** argv) {
   if (argc != 2) {
      printf("Usage: %s <number>\n", argv[0]);
      exit(0);
   }
   int f = atoi(argv[1]); /* f is a signed integer - when it gets value -1 */
   read_user_data(f);     /* its value is 0xFFFFFFFF. Cast as an unsigned  */
                          /* int it becomes a large positive number          */
}
```

Compile `type2.c` and run it like this:

```
[prompt]$ type2 0
length=0
will read 0 bytes
success
[prompt]$ type2 23
length=23
will read 23 bytes
success
[prompt]$ type2 56
length=56
not enough room in buffer
```

Exercise 3:

Create a variant of `type2.c`, call it `type2_1.c`, that removes `printf` statements and returns 0 if `length` is greater than 32 and otherwise returns the number of bytes `local_read` will read from the buffer `buf`. Create Cryptol functions `local_read`, `get_user_length`, `read_user_data` that perform like the corresponding C functions, except safely. Create a SAW file, type2.saw, that will be used to find an input to the C read_user_data that causes a huge read. Print the output of saw type2.saw.