Type Conversions

Type Conversions

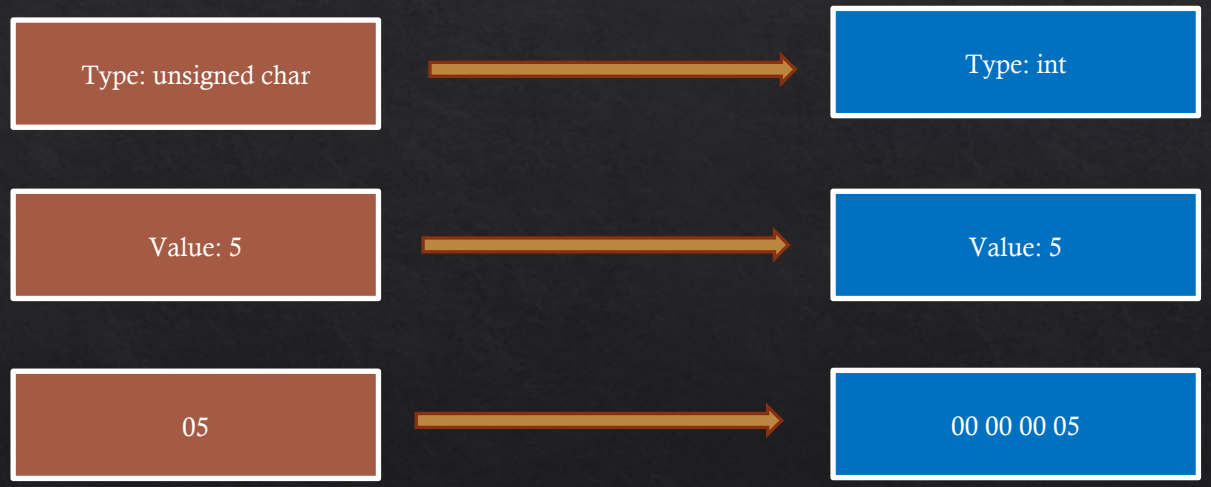
- Implicit (https://en.cppreference.com/w/c/language/conversion) and explicit
- Implicit – the compiler converts in the background to make the program function properly
- Explicit – the programmer casts, eg int i = (int)char_c;
- Conversion rules: simple conversions, integer promotions, arithmetic conversions

Conversion Rules for Integer Types

- Value preservation:
  - If the new type can represent all possible values of the old type, this conversion is “value preserving”
  - Widening casts are mostly value preserving
  - eg: char converted to int
- Value changing:
  - The new type cannot represent the old type
  - eg: int converted to char
  - Narrowing casts are all value changing

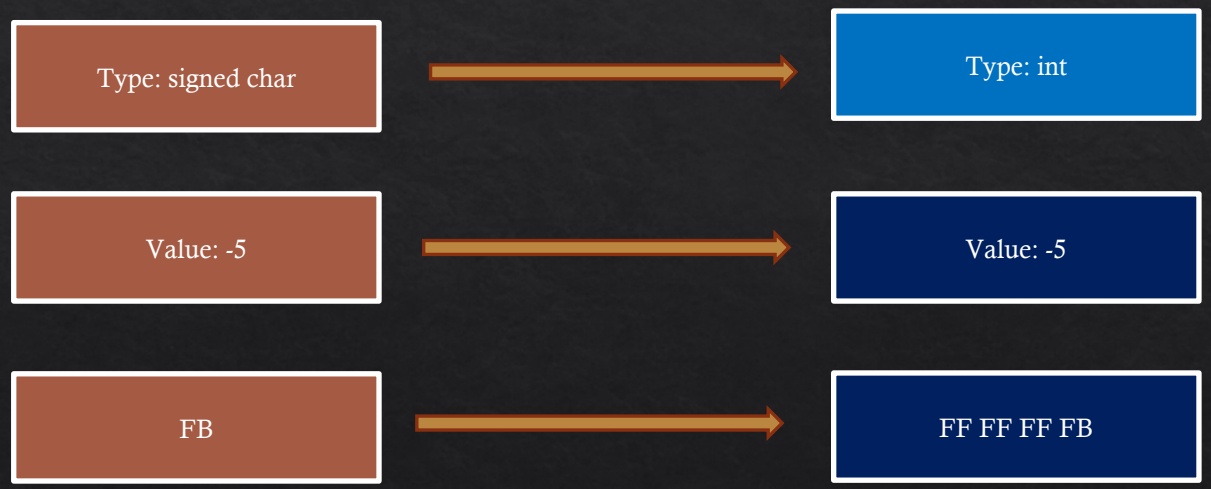
Value Preserving Conversion – Widening

- Bit pattern from old variable is copied to new variable
- If the old variable is **unsigned**, zero extension is used, which propagates 0 to all high bits



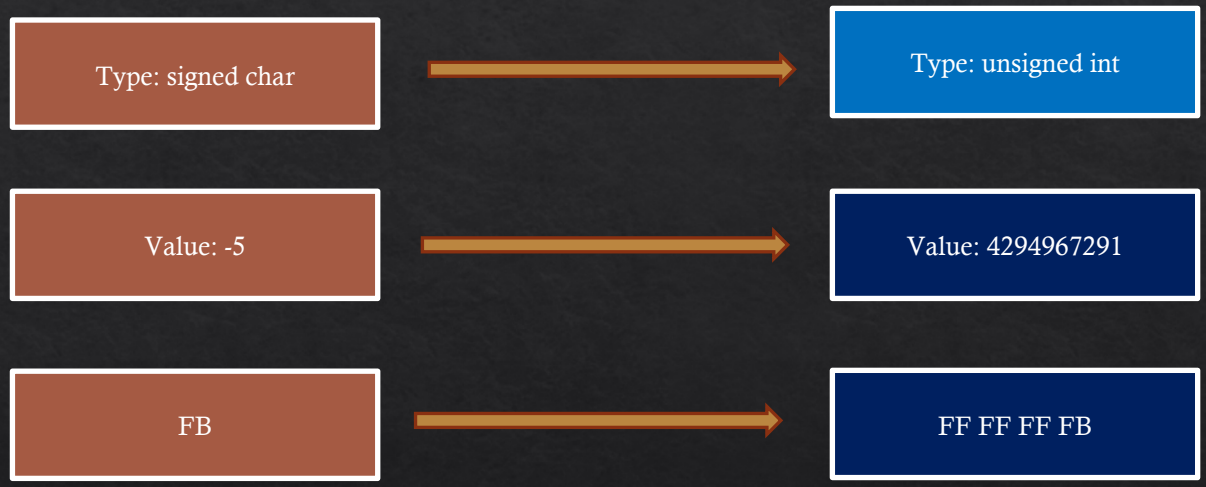
Value Preserving Conversion – Widening

- Bit pattern from old variable is copied to new variable
- If the old variable is **signed**, sign extension is used, which propagates the sign bit to all unused bits



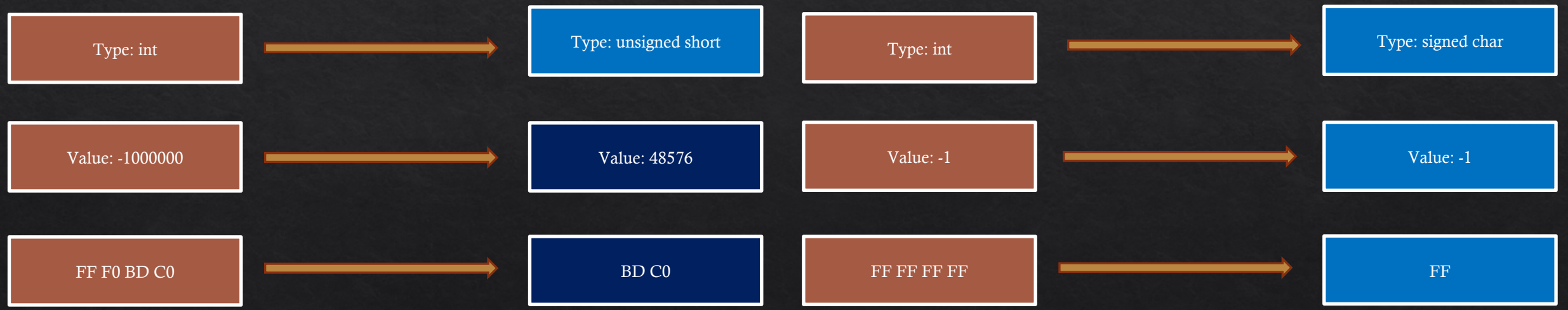
Value Changing Conversion – Widening

- If a signed char is converted to an unsigned int, this conversion will be a value changing conversion



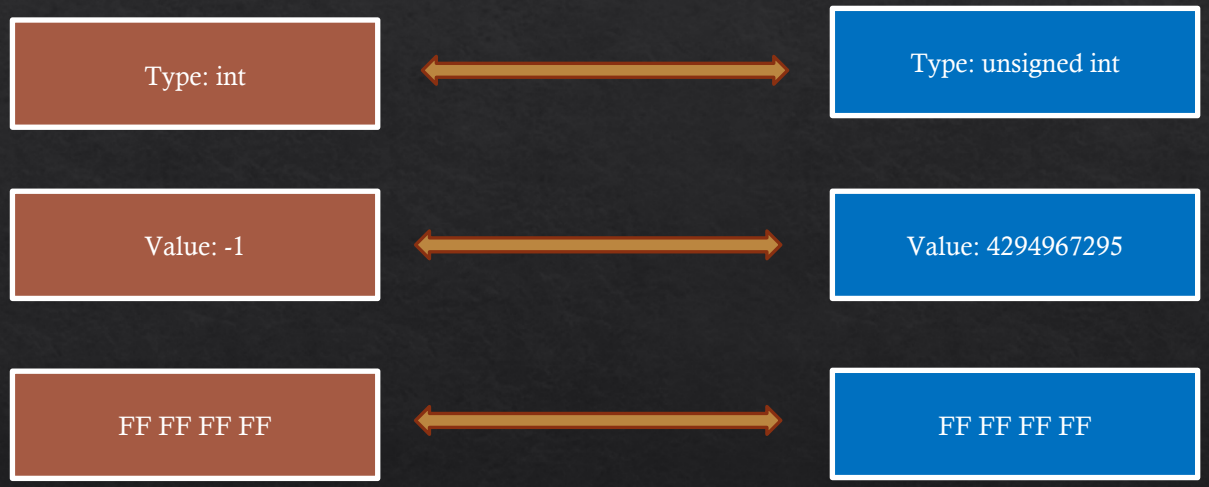
Narrowing

- Truncation
- Bits that don't fit into the new narrow type are dropped
- All narrowing conversions are value changing as precision is lost



Integer Conversion – Signed & Unsigned, Same Width

- Bit pattern does not change
- However it is now interpreted differently



Integer Conversion – Signed & Unsigned

- Result printed: A: 000000ff, 255, B: ffffffff, -1

```c
int main(void)
{
    unsigned char a = 0xff;
    char b = a;
    printf("A: %08x, %d, B: %08x, %d\n", a, a, b, b);
    return 0;
}
```

- Bit pattern doesn't change (b is still 0xFF), but is now interpreted as the sign bit (hence -1 instead of 255)
- https://cwe.mitre.org/data/definitions/195.html

Integer Conversion Rules Summary

	char	unsigned char	short int	unsigned short int	int	unsigned int	long	unsigned long
char		Value changing Same bits	Value preserving Sign extend	Value changing Sign extend	Value preserving Sign extend	Value changing Sign extend	Value preserving Sign extend	Value changing Sign extend
unsigned char	Value changing Same bits		Value preserving Zero extend	Value preserving Zero extend	Value preserving Zero extend	Value preserving Zero extend	Value preserving Zero extend	Value preserving Zero extend
short int	Value changing Truncation	Value changing Truncation		Value changing Same bits	Value preserving Sign extend	Value changing Sign extend	Value preserving Sign extend	Value changing Sign extend
unsigned short int	Value changing Truncation	Value changing Truncation	Value changing Same bits		Value preserving Zero extend	Value preserving Zero extend	Value preserving Zero extend	Value preserving Zero extend
int	Value changing Truncation	Value changing Truncation	Value changing Truncation	Value changing Truncation		Value changing Same bits	Value preserving Sign extend	Value changing Sign extend
unsigned int	Value changing Truncation	Value changing Truncation	Value changing Truncation	Value changing Truncation	Value changing Same bits		Value preserving Zero extend	Value preserving Zero extend
long	Value changing Truncation	Value changing Truncation	Value changing Truncation	Value changing Truncation	Value ?*	Value ?*		Value ?*
unsigned long	Value changing Truncation	Value changing Truncation	Value changing Truncation	Value changing Truncation	Value ?*	Value ?*	Value ?*	

*depends on sizeof(long) vs sizeof(int)
If sizeof(long) == sizeof(int), then value does not change
Else truncation for long → int, and sign extend/zero extension as required

Simple Conversions

- Casts, cast precedence – sum is first casted to a double before dividing by count

```
int sum = 22, count = 5;
double mean = (double)sum / count;
```

- Assignments – compiler converts b from int to short int, resulting in truncation

```
short int a;
int b = -1;
a = b;
```

Simple Conversions

- Function Calls: Prototypes – if the function has a prototype, types are converted using the Integer Conversion Rules table, else, default argument promotions kick in (next section: Integer Promotions)

```
int potato(int aa, unsigned char bb);

void f()
{
    char a=42;
    unsigned short b=42;
    long long int c = potato(a, b);
}
```

- Function Calls: return – converts the type of the variable to the type declared by the function. int a is converted to char

```
char f()
{
    int a=42;
    return a;
}
```

Integer Promotions

- What is it
  - **If an int can represent all values of the original type, the value is converted to an int; otherwise, it is converted to an unsigned int**
  - Responsible for taking a narrower integer type or bit field and promoting it to an integer or unsigned int
- When does it happen
  - Used by operators that require a specific operand type OR during arithmetic operations with different types. eg: b is promoted to int

```
int a=0; char b=1;
int c=a+b;
```

- How
  - Integer conversion rank, goes from smallest width to highest, signed/unsigned are at the same rank but unsigned takes precedence, only applies on integer-based data:
    - Long long int, unsigned long long int ≻ long int, unsigned long int ≻ unsigned int, int ≻ unsigned short, short ≻ char, unsigned char ≻ bool
- Why
  - Avoid arithmetic errors in intermediate values
- https://www.oreilly.com/library/view/c-in-a/0596006977/ch04.html
- http://www.idryman.org/blog/2012/11/21/integer-promotion/

Integer Promotion Applications

- Unary +

```
char a;
(+a) // the resulting type of this expression is int
(a) // the resulting type of this expression is char
```

- Unary –
  - Operand is promoted and negation is performed regardless if the operand is signed after promotion
- Unary ~
  - Operand is promoted and 1's complement is done
- Bitshift Operator
  - Both operands are promoted, the type of the result is the same as the promoted type of the left operand. a and c are promoted to integer, result of "a<<c" is promoted to integer

```
char a=1;
char c=16;
int bob;
bob=a << c;
```

Integer Promotion Applications

- Switch statements

```
switch (controlling expression)
{
    case (const int expr):
        body;
        break;
}
```

- Promotions are applied to the controlling expression and all integer constants are converted to the promoted control expression

Integer Promotion Applications

```c
#include <stdio.h>
int main()
{
    char a = 30, b = 40, c = 10;
    char d = (a * b) / c;
    printf ("%d ", d);
    return 0;
}
```

- a, b, and c are chars.
- a * b ➔ 30 * 40 = 1200 (that's greater than char size! Will this overflow?)
- No overflow because integer promotion was done on the operands – avoid arithmetic errors on intermediate values

Integer Promotion Applications

- Is a == b?

```c
#include <stdio.h>
int main()
{
    char a = 0xfb; // -5
    unsigned char b = 0xfb; // 251

    printf("a = %c", a);
    printf("\nb = %c", b);

    if (a == b)
      printf("\nSame");
    else
      printf("\nNot Same");
    return 0;
}
```

```
/tmp/Zq6XNcCHeT.o
a = 
b = 
Not Same
```

0xfb can be -5 or 251

Equality operation promotes both operands to int

As a is signed, when promoted, it becomes -5 (0xFFFB) (sign extension)
b is unsigned, so it is promoted as 251 (0x00FB) (zero extension)

Therefore a != b

Type Conversion Vulnerabilities

Type Conversion Vulnerabilities – Signed/Unsigned

- What are the issues here? read(int, void*, size_t)

```
int read_user_data(int sockfd)
{
    int length;
    char buffer[1024];

    length = get_user_length(sockfd); // user input
    if (length > 1024)
        return -1;

    if (read(sockfd, buffer, length) < 0)
        return -1;

    return 0;
}
```

1. no negative check for length
2. read(int, void*, size_t)
   negative length values will be converted to a large unsigned int that can cause buffer OOB reads

Look for signed/unsigned conversions, eg int types passed to functions that take in size_t or some unsigned int type
Eg: read(), recvfrom(), memcpy(), memset(), bcopy(), snprintf(), strncat(), strcpy(), malloc()
Check user input where the value might be interpreted as +ve or –ve in its usage lifetime

Type Conversion Vulnerabilities – Sign Extension

- Sign extension can be value changing

```
char len;
len = get_len_field();
snprintf(dst, len, "%s", src);
```

- snprintf(char*, size_t, char*, …)
- If len is negative, it will be converted to an unsigned int when used by snprintf
- Proposed fix, does it work?

```
char len;
len = get_len_field();
snprintf(dst, (unsigned int)len, "%s", src);
```

- Signed types converted to an unsigned type goes through sign extension, so if len is negative, converting it to unsigned will turn it into a large number, therefore fix doesn't work

```
1  // Online C compiler to run C program online
2  #include <stdio.h>
3  #include <limits.h>
4  int main(void)
5  {
6      char c = -10;
7      printf("%x, %u, %d\n",c,c,c,c);
8      return 0;
9  }
```

```
/tmp/l8irHFvPSd.o
fffffff6, 4294967286, -10
```

Case Study: Type Conversion Vulnerabilities – Antisniff

- https://www.informit.com/articles/article.aspx?p=686170&seqNum=6
- The vulnerable function (watch_dns_ptr) extracts the domain name from a packet and copies it into nameStr
- DNS packets represent domains like this (test.jim.com):

4	t	e	s	t	3	j	i	m	3	c	o	m	0

Case Study: Type Conversion Vulnerabilities – Antisniff v1.0

- So for input: 4test3jim3com0, this function tries to reassemble it to "test.jim.com"

count = 4

```
char *indx;
int count;
char nameStr[MAX_LEN]; //256
...
memset(nameStr, '\0', sizeof(nameStr));
...
indx = (char *)(pkt + rr_offset);
count = (char)*indx;

while (count)
{
    (char *)indx++;
    strncat(nameStr, (char *)indx, count);
    indx += count;
    count = (char)*indx;
    strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
}
nameStr[strlen(nameStr)-1] = '\0';
```

i = 1
strncat(nameStr, indx[i], 4)
i = 5
count = 3
strncat(nameStr, ".", 256 – 4);

What is the problem with this code?

No boundary check for count

count can be > 256 which will cause strncat to OOB write to nameStr

- After the loop is done, nameStr should contain test.jim.com\0

Case Study: Type Conversion Vulnerabilities – Antisniff v1.1

```
char *indx;
int count;
char nameStr[MAX_LEN]; //256
...
memset(nameStr, '\0', sizeof(nameStr));
...
indx = (char *)(pkt + rr_offset);
count = (char)*indx;

while (count)
{
    if (strlen(nameStr) + count < ( MAX_LEN - 1) )
    {
        (char *)indx++;
        strncat(nameStr, (char *)indx, count);
        indx += count;
        count = (char)*indx;
        strncat(nameStr, ".",
                sizeof(nameStr) - strlen(nameStr));
    }
    else
    {
        fprintf(stderr, "Alert! Someone is attempting "
                        "to send LONG DNS packets\n");
        count = 0;
    }
}
nameStr[strlen(nameStr)-1] = '\0';
```

Added check

What is the problem with this code?

strlen returns size_t
→ unsigned int + int

count will be converted to unsigned int

Therefore
if count is -1, and nameStr = "aaaaa"
strlen(nameStr) + count →
5 + 4294967295
which overflows unsigned int (2^{32}), and becomes 4, which passes the new check

strncat(char*, char*, size_t)
As count is -1, it will be converted to 4294967295 before it is passed into strncat

Case Study: Type Conversion Vulnerabilities – Antisniff v1.1.1

```
char *indx;
int count;
char nameStr[MAX_LEN]; //256
...
memset(nameStr, '\0', sizeof(nameStr));
...
indx = (char *)(pkt + rr_offset);
count = (char)*indx;

while (count)
{
    /* typecast the strlen so we aren't dependent on
    the call to be properly setting to unsigned. */
    if ((unsigned int)strlen(nameStr) + (unsigned int)count < ( MAX_LEN - 1) )
    {
        (char *)indx++;
        strncat(nameStr, (char *)indx, count);
        indx += count;
        count = (char)*indx;
        strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
    }
    else
    {
        fprintf(stderr, "Alert! Someone is attempting "
                        "to send LONG DNS packets\n");
        count = 0;
    }
}
nameStr[strlen(nameStr)-1] = '\0';
```

Added casts

What is the problem with this code?

strlen returns size_t (unsigned int)
So the explicit cast is superfluous

count will automatically be converted to unsigned int due to the type of size_t, so this is superfluous

Therefore, this fix does the same thing as the previous version!

Case Study: Type Conversion Vulnerabilities – Antisniff v1.1.2

Changed from signed to unsigned

indx is an unsigned char pointer

Dereferencing it gives an unsigned char, which is then casted to a char

This char gets casted to unsigned int on assignment to count

Therefore… if indx = 0xFB (251), it gets casted to char, thus 0xFB → -5 (bit pattern doesn't change)

Then it gets casted to unsigned int (when assigned to count). This conversion results in sign extension, so 0xFFFFFFFB → large value(!)

```
unsigned char *indx;
unsigned int count;
unsigned char nameStr[MAX_LEN]; //256
...
memset(nameStr, '\0', sizeof(nameStr));
...
indx = (char *)(pkt + rr_offset);
count = (char)*indx;

while (count)
{
    if (strlen(nameStr) + count < ( MAX_LEN - 1) )
    {
        indx++;
        strncat(nameStr, indx, count);
        indx += count;
        count = *indx;
        strncat(nameStr, ".",
                sizeof(nameStr) - strlen(nameStr));
    }
    else
    {
        fprintf(stderr, "Alert! Someone is attempting "
                        "to send LONG DNS packets\n");
        count = 0;
    }
}
nameStr[strlen(nameStr)-1] = '\0';
```

But… it solves the bug because

At first iteration, strlen(nameStr) == 0, therefore 0xFFFFFFFB < MAX_LEN – 1 will go to the fail case

At subsequent iterations, count = *indx which is unsigned char being assigned to unsigned int, which is value preserving

Subtle bug, if the while loop never runs, nameStr[-1] will be written with a null

Case Study: Type Conversion – Sign Extension

- https://www.informit.com/articles/article.aspx?p=686170&seqNum=6
- Focus on code that handles signed character values or pointers or signed short integer values or pointers
- Usually found in string handling and packet decoding network code
- Look for code that takes in a char or short and uses it in a context that causes it to be turned into an integer
- Look for the movsx instruction – move with sign extension

Case Study: Type Conversion – Sign Extension

read_length returns unsigned short

- What is the problem here?

```c
unsigned short read_length(int sockfd)
{
    unsigned short len;
    if(full_read(sockfd, (void *)&len, 2) != 2)
        die("could not read length!\n");
    return ntohs(len);
}

int read_packet(int sockfd)
{
    struct header hdr;
    short length;
    char *buffer;

    length = read_length(sockfd);
    if(length > 1024) {
        error("read_packet: length too large: %d\n", length);
        return -1;
    }

    buffer = (char *)malloc(length+1);
    if((n = read(sockfd, buffer, length) < 0) {
        error("read: %m");
        free(buffer);
        return -1;
    }

    buffer[n] = '\0';
    return 0;
}
```

Unsigned short assigned to short
If read_length returns 0xFFFF (65535)
length will be assigned to 0xFFFF (-1)

length is promoted to int (sign extension) → 0xFFFFFFFF (2^{32}-1)

The length check will be passed

length is promoted to int (sign extension) → 0xFFFFFFFF before adding 1
Therefore malloc will receive a small value (overflow), thus 0 in this case

read(int, void*, size_t)
length is promoted to int (sign extension) → 0xFFFFFFFF

read will be called with a large length value, causing buffer overflow

Truncation

Truncation

- Wider types will be truncated when converted to narrower types

```
int g = 0x12345678;
short int h;
h = g;
```

- h will contain 0x5678 as the higher 2 bytes are truncated
- Look for locations where shorter data types are used to track values or used to hold a calculation result
- Look in structure definitions, especially in network oriented code

Truncation

- What is the problem here?

seteuid(uid_t)
setuid(uid_t)

uid_t is an unsigned int

What if the MAX_SHORT +1 (65536) value is passed in?

It will pass the check since 65536 != 0

When it gets passed into assume_privs, int will get truncated to unsigned short

0xFFFF + 0x1 = 0x00 10 00 00

The higher byte will be dropped, therefore assume_privs will receive 0

```
void assume_privs(unsigned short uid)
{
    seteuid(uid);
    setuid(uid);
}
int become_user(int uid)
{
    if (uid == 0)
        die("root isnt allowed");

    assume_privs(uid);
}
```

Case Study: Truncation – Linux Kernel LPE via Filesystem

- https://blog.qualys.com/vulnerabilities-threat-research/2021/07/20/sequoia-a-local-privilege-escalation-vulnerability-in-linuxs-filesystem-layer-cve-2021-33909
- https://news.ycombinator.com/item?id=27893181
- Root cause: size_t → int conversion

Case Study: Truncation – Linux Kernel LPE via Filesystem

```
ssize_t seq_read_iter(struct kiocb *iocb, struct iov_iter *iter)
{
        struct seq_file m = iocb->ki_filp->private_data;
        /* grab buffer if we didn't have one */
        if (!m->buf) {
                m->buf = seq_buf_alloc(m->size = PAGE_SIZE);
        }
        // get a non-empty record in the buffer
        while (1) {
                err = m->op->show(m, p);
                if (!seq_has_overflowed(m)) // got it
                        goto Fill;
                // need a bigger buffer
                kvfree(m->buf);
                m->buf = seq_buf_alloc(m->size <<= 1);
        }
}
```

m->size is doubled
Remember this line

Case Study: Truncation – Linux Kernel LPE via Filesystem

If an attacker creates+mounts+deletes a dir with path length of > 1GB

show_mountinfo → seq_dentry
Which retrieves buffer size of size_t (2GB)

seq_dentry → dentry_path
size_t size is casted to int buflen

unsigned int size → int buflen
size is 2147483648
buflen's int max value is 2147483647

Overflow, so buflen is –INT_MAX

Thus p = -2GB below buf

dentry_path → prepend
buflen is decreased by 10, so it becomes a large positive value
buffer is decreased by 10 and points to an offset of -2GB - 10b

The string "//deleted" is written to an offset of -2GB – 10b and is OOB

```
---------------------------------------------------------------------
static int show_mountinfo(struct seq_file *m, struct vfsmount *mnt)
{
...
        seq_dentry(m, mnt->mnt_root, " \t\n");
---------------------------------------------------------------------
int seq_dentry(struct seq_file *m, struct dentry *dentry, const char *esc)
{
        char *buf;
        size_t size = seq_get_buf(m, &buf);
…
        if (size) {
                char *p = dentry_path(dentry, buf, size);
---------------------------------------------------------------------
char *dentry_path(struct dentry *dentry, char *buf, int buflen)
{
        char *p = NULL;
…
        if (d_unlinked(dentry)) {
                p = buf + buflen;
                if (prepend(&p, &buflen, "//deleted", ) != 0)
---------------------------------------------------------------------
static int prepend(char **buffer, int *buflen, const char *str, int namelen)
{
        *buflen -= namelen;
        if (*buflen < 0)
                return -ENAMETOOLONG;
        *buffer -= namelen;
        memcpy(*buffer, str, namelen);
---------------------------------------------------------------------
```