Overview

• Common vulnerabilities in C programs.
• What are the causes for vulnerabilities.
• What safety guarantees would one like.
• How can a type-system provide these guarantees.
Buffer Overflows

```c
foo()
{
    char [10] array;
    ...
    strcpy(array, input);
    ...
    return;
}
```
Why do buffer overflows happen?

- C-arrays are not the abstract data types that programmers think.
- Arrays are implemented using pointers.
- Pointer arithmetic can violate array abstraction.
Fat pointers

• Arrays are represented by a special type of pointer, written as <type>?.

• The pointer records the bounds information along with the address, hence the name.

• Pointer arithmetic is allowed on fat pointers.
“Safe” strcpy

```c
char *strcpy(char *dest, char *src){
    int i, len_d, len_s;
    len_d = dest.size;
    len_s = src.size;
    for(i=0; i < min(len_d, len_s) - 1; i++){
        *(dest+i) = *(src+i);
    }
    *(dest+i) = NULL;
    return dest;
}
```
Fat pointers (contd.)

- Compiler converts arrays and strings into fat pointers automatically.
- A fat pointer can be cast into a normal pointer. This incurs a runtime bounds check.
- Normal pointers do NOT allow pointer arithmetic.
- A normal pointer can be cast into a fat pointer of size 1.
Uninitialized pointers

```c
foo()
{
    FILE *f;
    ...
    x = getc(f);
}
```
Uninitialized pointers

foo()
{
    FILE *f;
    ...
    x = getc(f);
}
Avoiding NULL pointer dereference

- Possibility 1: Definite initialization
  - Require that all pointers be initialized before use.
  - Can only be conservatively checked by static analysis.
Avoiding NULL pointer dereference

- Possibility 2: Have a kind of pointer that can never be NULL, written as `<type>@`.
- Creating a pointer of this kind requires a NULL check in the beginning.
- The type system ensures that the pointer can never be made NULL.
- No need for further checks.
getc with non-NULL pointers

```c
foo(){
    FILE *f;
    f = fopen("/afs/abc.txt");
    FILE* f' = (FILE*)f;
    while(inp != EOL){
        inp = getc(f');
    }
}
```
getc with non-NULL pointers

foo()
{
    FILE *f;
    f = fopen("/afs/abc.txt");
    FILE@ f' = (FILE@)f;
    while(inp != EOL){
        inp = getc(f');
    }
}

getc: FILE@ → int
getc with non-NULL pointers

```c
foo()
{
    FILE *f;
    f = fopen("/afs/abc.txt");
    FILE f' = (FILE)f;
    while(inp != EOL){
        inp = getc(f');
    }
}
```

```c
getc : FILE* → int
```
putc with non-NULL pointers

```c
foo()
{
    FILE *f;
    int inp = getc(f);
}

getc : FILE@ → int
```
Dangling pointers

```c
char *itoa(int i){
    char buf[20];
    sprintf(buf, "%d", i);
    return buf;
}
```

buf is a local variable allocated on stack
buf would no longer be available upon return

Need to check if the region of memory in which ‘buf’ lives, is still active after return.
Regions for memory management

- Memory is organized as regions.
- A region is a conceptual container for all objects.
- Regions can grow and die.
- In order to read/write a pointer, its region must be alive.
- This is ensured statically, by typechecking.
Examples of regions

- A heap is a region, with lifetime equal to that of the whole program.
- A stack frame is a region, which dies upon return from the procedure.
- Explicit regions can also be declared using the `region` construct.
Explicit regions

region h {
    int@h x = rmalloc(h, sizeof(int));
    int?@h y = rnew(h) { 1, 2, 3 };
}

Special kind of malloc for allocating in a region

Arrays initialized

Notice that pointer types are annotated with regions

No free()!
All the memory in a region is reclaimed at once when the region dies
Static region analysis

- The aim is to keep track of:
  - what regions are active at a point in the computation.
  - what region each pointer belongs to.
A type system for region analysis

- Set of region variables
- Type assignment to variables
- Constraints on region liveness
- Set of live regions
- Statement being typed

Constraints

\[ \rho_1 \leq \rho_2 \]
More judgments

\[ \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} e : \tau \]

Used to type expressions on rhs/lhs of \( = \)

\[ \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{lhs}} e : \tau@\rho \]

Tests if the type \( \tau \) makes sense in the context \( \Delta \)

\[ \Delta \vdash \tau \text{ type} \]
char *say_hello(){
    char buf[20];
    return strcpy(buf, "Hello");
}

char *@hp say_hello(){
    char *@sh buf= rmalloc(sh, sizeof(char) * 20);
    return strcpy(buf, "Hello");
}

Associated with each function is a unique region
Let **sh** be the stack frame region of say_hello.
Let **hp** be the region of the heap.
Γ = strcpy : char * @ρ × const char * →^{hp, ρ} char * @ρ

Δ = hp
\[ \Gamma = \text{strcpy} : \text{char}^* \times \text{const char}^* \rightarrow^{hp, \rho} \text{char}^* \times \text{@\rho} \]

\[ \Delta = hp \]

\[ \Delta, sh \]
\[ \Gamma = \text{strcpy} : \text{char} \times \text{const char} \to \text{hp, char} \backslash \text{sh} \]

\[ \Delta = \text{hp} \]

\[ \Delta, \text{sh} ; \Gamma, \text{buf} : \text{char} \backslash \text{sh} \]
\[ \Gamma = \texttt{strcpy} : \texttt{char} \times @\rho \times \texttt{const char}^* \rightarrow^{hp, \rho} \texttt{char} \times @\rho \]

\[ \Delta = hp \]

\[ \Delta, sh ; \Gamma, \texttt{buf} : \texttt{char} \times @sh ; \gamma, \epsilon \leq sh \]
\[
\Gamma = \text{strcpy} : \text{char}^{*} \times \text{const char}^{*} \rightarrow \text{hp}, \rho \text{ char}^{*} \times \rho \\
\Delta = \text{hp}
\]

\[
\Delta, sh ; \Gamma, \text{buf} : \text{char}^{*} \times sh ; \gamma, \epsilon \leq sh ; \epsilon \cup sh
\]
\[ \Gamma = \text{strcpy} : \text{char} \ast \@\rho \times \text{const char} \ast \to^{hp,\rho} \text{char} \ast \@\rho \]

\[ \Delta = hp \]

\[ \Delta, sh ; \Gamma, \text{buf} : \text{char} \ast \@sh ; \gamma, \epsilon \leq sh ; \epsilon \cup sh ; \text{char} \ast \@\rho \]
$\Gamma = \text{strcpy} : \text{char} \ast @\rho \times \text{const char} \ast \rightarrow ^{hp, \rho} \text{char} \ast @\rho$

$\Delta = hp$

$\Delta, sh ; \Gamma, buf : \text{char} \ast @sh ; \gamma, \varepsilon \leq sh ; \varepsilon \cup sh ; \text{char} \ast @\rho \models \text{return strcpy}(buf, \text{"Hello"})$
\[ \Gamma = \text{strcpy} : \text{char}^* \oplus \rho \times \text{const char}^* \rightarrow^{hp, \rho} \text{char}^* \oplus \rho \]

\[ \Delta = hp \]

\[ C \vdash_{\text{rhs}} \text{strcpy}(\text{buf}, \text{"Hello"}) : \text{char}^* \oplus \rho \]

\[ \Delta, sh ; \Gamma, \text{buf} : \text{char}^* \oplus sh ; \gamma, \epsilon \leq sh ; \epsilon \cup sh ; \text{char}^* \oplus \rho \vdash \text{return strcpy}(\text{buf}, \text{"Hello"}) \]
\[ \Gamma = \text{strcpy} : \text{char} \times \Rightarrow \text{const char} \rightarrow \text{char} \times \Rightarrow \text{const char} \]

\[ \Delta = \text{char} \times \Rightarrow \text{const char} \]

But \[ C \vdash_{\text{rhs}} \text{strcpy}(\text{buf}, \text{"Hello"}) : \text{char} \times \Rightarrow \text{sh} \]

\[ C \vdash_{\text{rhs}} \text{strcpy}(\text{buf}, \text{"Hello"}) : \text{char} \times \Rightarrow \rho \]

\[ \Delta, \text{sh} ; \Gamma, \text{buf} : \text{char} \times \Rightarrow \text{sh} ; \gamma, \epsilon \leq \text{sh} ; \epsilon \cup \text{sh} ; \text{char} \times \Rightarrow \rho \vdash \text{return strcpy}(\text{buf}, \text{"Hello"}) \]
\[ \Delta, sh ; \Gamma, buf : \text{char} \ast \mathbin{@sh} ; \gamma, \epsilon \leq sh ; \epsilon \cup sh ; \text{char} \ast \mathbin{@sh} \vdash \text{return strcpy(buf, "Hello")} \]

\[ \Delta; \Gamma; \gamma; \epsilon' \vdash \text{say_hello} : \text{unit} \rightarrow^\gamma \text{char} \ast \mathbin{@sh} \]
\[ \Delta \vdash \text{unit} \rightarrow \text{char} \ast \text{@sh} \ 	ext{type} \]

\[ \Delta, sh; \Gamma, \text{buf} : \text{char} \ast \text{@sh}; \gamma, \epsilon \leq sh; \epsilon \cup sh; \text{char} \ast \text{@sh} \vdash \text{return strcpy(} \text{buf, "Hello"}) \]

\[ \Delta; \Gamma; \gamma; \epsilon' \vdash \text{say
d_hello} : \text{unit} \rightarrow \gamma \text{ char} \ast \text{@sh} \]
\[ \Delta \vdash \text{unit} \rightarrow \text{char} \ast \mathbb{sh} \text{ type} \]

\[ \Delta, sh; \Gamma, \text{buf} : \text{char} \ast \mathbb{sh}; \gamma, \epsilon \leq sh; \epsilon \cup sh; \text{char} \ast \mathbb{sh} \vdash \text{return strcpy}(\text{buf}, \text{"Hello")}) \]

\[ \Delta; \Gamma; \gamma; \epsilon' \vdash \text{say_hello} : \text{unit} \rightarrow \gamma \text{ char} \ast \mathbb{sh} \]
Typing Region declaration

\[
\Delta \vdash \tau \text{ type} \quad \Delta, \rho; \Gamma; \gamma, \epsilon \leq \rho; \epsilon \cup \rho; \tau \vdash s \\
\Delta; \Gamma; \gamma; \epsilon; \tau \vdash \text{region}(\rho)\{s\}
\]
Conclusion

• Type systems provide a means to build languages with strong static guarantees like memory safety, control flow safety.

• In places where static analysis does not suffice, a combination of static and runtime checks might be helpful.
But is memory safety enough?

```c
sudo(command cmd, string pass){
    /* check password */
    Hash h = crypt(pass);
    FILE *f = fopen("/etc/shadow");
    /* match password hashes */
    if(strcmp(..., h)){
        ...
        print(pass);
        ...
    }
}
```