C5 Other vulnerabilities

1. Evil Inputs

1.1 Environment Variables

- Variables kept by the shell for configuring the behaviour of utility programs

```
PATH                # The search path for shell commands (bash)
TERM                # The terminal type (bash and csh)
DISPLAY             # X11 - the name of the display
LD_LIBRARY_PATH     # Path to search for object and shared libraries
HOSTNAME            # Name of this UNIX host
PRINTER             # Default printer (lpr)
HOME                # The path to the home directory (bash)
home                # The path to the home directory (csh)
PS1                 # The default prompt for bash
prompt              # The default prompt for csh
```

Inheritance

- Inherited by default from the parent process
- Caller can set env variables for the program called to potentially dangerous values
- Internally stored as a pointer to an array of pointers to characters, terminated by a NULL pointer
- Multiple entries with the same variable name but different values can exist
    - Program can check for one of these values but use a different one

Time of Check to Time of Use

- A program that iterates across env variables might use a later matching entry instead of the first, which makes attacks possible if a check is made against the first matching entry but the value used is a later matched one

Internal Field Separator

- Determines how bash recognises fields or word boundaries when interpreting character strings, default value is `<space><tab><newline>`
- Consider an application which checks that user input does not contain one of the standard field separators
- An attacker who can set the IFS environment variable may be able to bypass the check

- Example

  - Example: sending email in C
    - `system("mail");` // this is equivalent to `sh -c mail`
    - A program with root permission would be tricked: attacker put his own "`mail`" executable in some directory which is then put in the front of PATH
  - Fix: use full pathname
    - `system("/bin/mail");` // or "`/usr/bin/mail`"
    - Creative attacker adds the slash character "/" to IFS, and it becomes:
      `bin mail`
    - This time a custom executable called "bin" will do the trick
    - This has been fixed by making the shells not inherit the IFS variable

Bof via Env variables

- Buffer overflow can happen when changing env variables

```
int main(void)
{
  char *ptr_h;
  char h[64];
  ptr_h = getenv("HOME");
  if(ptr_h != NULL) {
    sprintf(h, "Your home directory is: %s !", ptr_h);
    printf("%s\n", h);
  }
  return 0;
}
```

- **Defence**
  - Extract list of environment variables needed as input
  - Erase entire environment using `clearenv()` or set environment with `env` using `-i` option
  - Set Safe values for the environment variables needed
    - IFS, Timezone
    - Do not include current directory in `PATH`

1.2 Callee-saved registers

Spilling Register Values to Stack

- Compiler may choose to fetch instructions from CPU registers rather than from memory, tracks which registers are currently in use and which registers it can assign new values to.
- If all registers are in use and one of the registers is required to perform a computation, then the compiler temporarily saves the content of the register to the stack

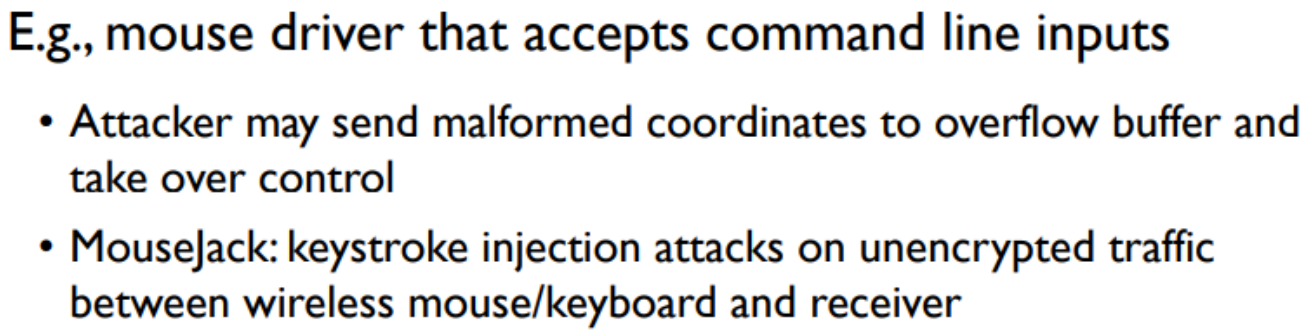
- When a function (caller) calls another function (callee), the callee does not know which of the registers are being used, and the caller expects the registers to hold the same values when callee returns control to the caller
- Callee saves the values of the registers (callee saved registers) it uses during its execution temporarily in its stack frame. Callee will then restore all callee saved registers before returning to the caller
    - Attacker who can change values on the stack can corrupt the callee saved registers

1.3 Symbolic links

- File that points to another file
- Access rights on the link may not be the same as access rights on the file
- Programs may change permissions for the file to that of the symbolic link
    - Attacker may create a symbolic link to a protected resource and then use this method to access the resource

1.4 Device drivers

- Device drivers are often third party products and usually run in kernel mode to perform low-level string operations
- Good targets for attackers



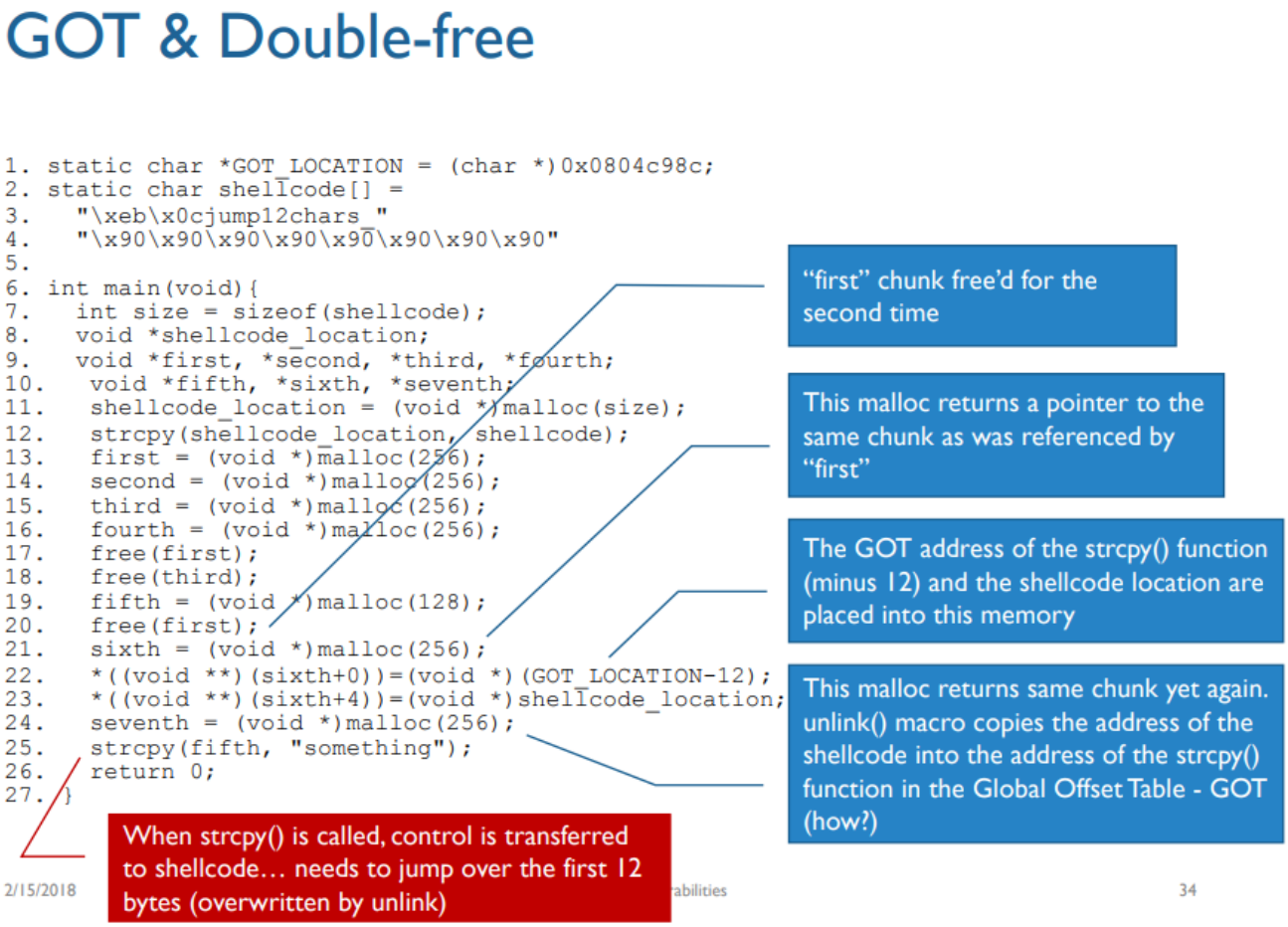
2. Targets to Overwrite

- Attacker must overwrite a parameter (e.g. return addresses) that **influences control or data flow** to take control of execution
    - `argv[]` method: stores shellcode in argument passed into program, requires an executable stack
    - ret2lbc method: calls system library; change to control flow but no shellcode inserted
- Address of a library function in Global Offset Table takes control when the function is called
- Tables provide a layer of indirection
    - When an 'item' (function, object, table, etc) is allocated dynamically in memory, its absolute address is not known at the time code is written

- Item will be referred to by an index in a table instantiated at runtime that contains the absolute address
- Attackers can insert an address into this table that directs to executable shellcode in memory

2.1 GOT

- A private table for a process (1:1 link) which holds absolute addresses of global variables / functions
  - Located at fixed offset from the module which is not known until module is linked
  - When code calls a function, it will go lookup the Procedure Linkage Table at index corresponding to the function. If the function is called for the first time, PLT will direct to GOT corresponding index.
  - GOT's corresponding index will by default point back to the PLT's resolver to resolve the absolute address of function and update GOT's index
  - In the following function calls of the same function, after going to PLT and then GOT, we can see that the function gets directly executed
- Sample attack



- Prepare attack:
  - Line 1: GOT_LOCATION is address of `strcpy()` in GOT
  - Lines 2-4: static array shellcode[] holds the shellcode
  - Lines 11-12: shellcode copied into a chunk
- Heap Feng Shui; *might a lookaside buffer hold the chunk most recently freed?*
  - Lines 13-16: allocate four chunks of size 256
  - Line 17: free first chunk (might be put into the lookaside buffer, but not yet into a bin)
  - Line 18: free third chunk (would push first chunk into a bin)
  - Line 19: allocate chunk of size 128 (might use space of third chunk, clears lookaside)
- Execute attack
  - Line 20: free first chunk again (corrupts the double-linked list in position of first chunk)
  - Line 21: allocate chunk of size 256 (taken from bin; positioned at the same memory address as the first chunk before; a free chunk at that address remains in the bin)
  - Line 22-23: write GOT_LOCATION and shellcode_location at offsets 0 and 4 into sixth chunk, i.e., in the position of forward and backward pointer in free chunk at that address
  - Line 24: allocate chunk of size 256; gets again the address given before to the first chunk; when the free chunk at that address is unlinked, the GOT is corrupted
  - Line 25: call to `strcpy()` will be redirected via the GOT to the shellcode

2.2 Address of exception handler

2.3 Pointers to Temporary Files

- A program that defines a temporary file and contains a buffer overrun vulnerability can be supplied a user input such that pointer to temporary file is overwritten with a pointer to a target file

2.4 Function Pointers

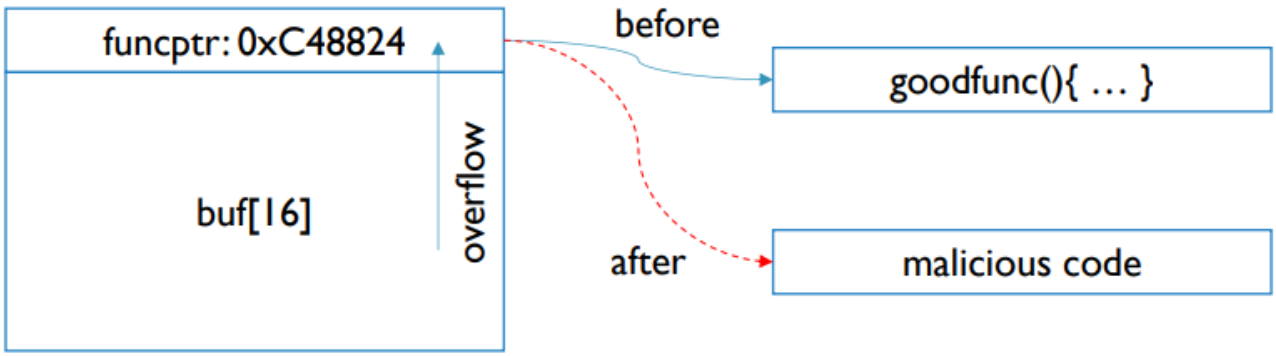
- Function pointer `int (*funcptr) (char *str)` allows dynamic modification of the function to be called. This means attacker can redirect execution by overwriting function pointer with location of first shellcode instruction

- Function pointers are control data so DEP does not stop usage of modified function pointers

```
int main(int argc, char **argv)
{ static char buf[16]; /* in BSS */
  static int (*funcptr)(const char *str); /* in BSS */
  funcptr = (int (*)(const char *str))goodfunc;

  /* We can cause buffer overflow here */
  strncpy(buf, argv[1], strlen(argv[1]));

  (void)(*funcptr)(argv[2]);
  return 0;
}
```



Virtual Table

- CZ2002: `virtual` allows functions to be overwritten
    - **Virtual**
        - To force method evaluation to be based on object type rather than reference type. [<ref type> <name> = new <obj type>(..)]
        - Without **virtual** => non polymorphic (no dynamic binding)
        - **Example : virtual** void area() { cout << "……." << endl ; }
        - **Virtual** function magic only operates on pointers(*) and references(&).
        - If a method is declared **virtual** in a class, it is **automatically virtual** in all **derived** classes.
    - **Pure method => abstract** method (pure virtual)
        - By placing **"= 0"** in its declaration
        - **Example : virtual** void area() **= 0** ; // abstract method
        - **The class becomes an abstract** class

- Lookup table for resolving function calls
- Static array containing one entry for each virtual function that can be called by objects of the class, each entry is a function pointer pointing to the most derived function accessible by that class
- Each class that uses virtual functions will have their own virtual table

2.5 Address of a destructor function

- Attacker can overwrite function pointers in the `.dtors` section for executables generated by GCC

- Attribute specifications include a constructor and a destructor, which are called before `main()` and after `main()` or when `exit()` is called respectively
- Addresses for constructors and destructors are stored in `.ctors` and `.dtors` sections in the generated ELF executable image (**writable by default**)

.dtors table

- First entry `0xFFFFFFFF` last entry `0x00000000`, addresses located between both entries
    - For modules without a `.dtors` table, there is no space between both entries
- Attacker can write shellcode address to the location **above** the address holding `0xFFFFFFFF` in the table. However, when destructors are not called when expected, and shellcode is executed instead, program may crash. So attackers tend to find modules that do not have a destructor (e.g. `printf()`)

2.6 Memory positions holding privileges

3. Memory Residues

- When a new process becomes active, it gets access to resources (e.g. memory positions) used by the previous process. Data left behind by the previous process is known as **storage residues**
- New processes using storage residues can cause undefined behaviour
- Storage residues can contain sensitive data, so OS's will only allow a process to read from positions it has written to before

3.1 Sun tarball

3.2 Heartbleed