

Aufgabe 1.

a)

There are 4 fundamental differences between a process and a thread:

- A process is a program in execution, and a thread is the minimal executable unit in which a process can be divided, that results in processes being heavier than threads.
- Processes require separate address space and threads share the same one.
- Inter process communication is expensive (address change) compared to the inter thread communication.
- Context switch from one process to another is also quite expensive in comparison the context switch between threads. wuz ✓

b)

Process Identifier: Number used by most OS kernels to uniquely identify and active process and refer to it. It is very relevant when identifying which process is using which I/O devices, or memory areas.

CPU states are operating modes/contexts that place restrictions to the operations that can be executed by processes in the CPU. During a context switch, the running process is stopped and another process is given a chance to run. Without modes, any process could execute any operation, leading to security breaches and possible system crashes when used by standard users.

Process Control Information is used by the OS to manage the process itself and it's divided into:

- Process scheduling state, which controls states of execution ("suspending", "waiting", etc) and priority levels of the processes.
- Process structuring information, in charge of the process identifiers from parent and children processes, or any other related to the running one in a functional way.
- Interprocess communication information, in charge of the flags, signals and messages associated with the communication among different processes.
- Process privileges to the system resources.
- Accounting information, which such data as how long a process took to execute, when was it last run, etc. ✓

c)

Program Counter: Also known as Instruction Pointer, it is a processor register used to indicate in which program sequence a computer is. It normally increases as one instruction was fetched and holds the memory address of the next instruction to be executed. wuz

Stack Pointer: It is a register used to indicate the location of the last item pushed into the stack. When something is pushed, the SP decrements before this has been done, and when something is popped (taken out of the stack), it increments. ✓

d)

Call Stack is a dynamic data structure maintained inside of the RAM by the operating system. Its purpose is to control the way procedures and functions call and pass parameters to each other. A Call Stack is maintained for each task and indeed for each thread. Below is how it works:

When a Procedure1 calls a Procedure2:

- (1) Any parameters that P2 is expecting are pushed into the call stack in the reversed order in which they were declared in its parameter list,
- (2) next the return address from P1 is also pushed into the stack (this address comes from the program counter and will be crucial later to return control back to P1, when P2 is finished)
- (3) Now we have P2 in scope, so we push its variables into the stack. P2 is now in control and its Stack frame -this one includes the variables, return address of calling procedure P1 and parameters of actual procedure- is in place, so it can get

ahold of any data it needs.

Would P2 call another procedure PX, steps (1),(2) and (3) would be repeated until PX is in control and has its Stack frame in place.

When PX is finished, PX variables are popped out of the stack and it returns control to P2 thanks to the return address in the stack. This will be reapeated until all procedures are finished and the Stack Call as 'clean' as at the beginning.

e)

(1) Welche Datenstruktur in C soll ausgenutzt werden?

Der Stack, auch call stack (Aufrufstapel) oder einfach "the stack" genannt, ist ein besonderer Stapspeicher, in dem zur Laufzeit Informationen von verwendeten Subroutinen gespeichert werden.

(2) Was passiert im Codebeispiel example2.c? Gegeben Sie ggf. eine Zeichnung an.

Zuerst wird ein Char Array "large\_string" mit 256 Byte initialisiert und in einem for-Loop komplett mit 'A's gefüllt. Danach wird die Funktion "function" aufgerufen und der Pointer auf "large\_string" mit übergeben. In "function" wird jetzt ein Char Array "buffer" mit 16 Byte initialisiert und mittelst von strcpy wird das 256 Byte große Array "large\_string" auf das 16 Byte große Array "buffer" kopiert. Wenn der Code ausgeführt wird, sieht der Stack nach dem function() Aufruf folgendermaßen aus:

```
buffer  
[ | | | | | | | | | | | | ]  
Saved Frame Pointer (SFP)  
[ | | | ]  
Return Adresse (RET)  
[ | | | ]  
*str  
[ | | | ]
```

Wenn jetzt die 256 Byte des "large\_string" Arrays auf "buffer" kopiert werden sollen, werden über die Array Grenze hinweg auch der folgende Speicher überschrieben (da "bottom of memory" = "top of stack"). Dabei wird die Return Adresse mit auch mit 'A's (hex: 0x41) überschrieben und lautet jetzt 0x41414141. Diese Adresse liegt außerhalb des dem Prozess zugewiesenen Adressbereichs und verursacht beim Rücksprung nach Ende der Funktion eine segmentation violation.

```
buffer  
[A|A|A|A|A|A|A|A|A|A|A|A|A|A|A|A]  
Saved Frame Pointer (SFP)  
[A|A|A|A]  
Return Adresse (RET)  
[A|A|A|A]  
*str  
[A|A|A|A]
```

(3) Welches Sicherheitsproblem wird durch das Codebeispiel example2.c deutlich?

Es kann durch geschickte Manipulation des Stacks erreicht werden, dass die Rücksprung Adresse mit einer anderen überschrieben wird und damit kann fremder Code ausgeführt werden.

(4) Was kann der Programmierer dagegen beim Programmieren tun?

Überprüfungen einbauen, die verhindern, dass zu große Datenmengen in einen zu kleinen Speicher (Buffer) geschrieben werden.

Bei typsicheren Programmiersprachen kann der Compiler oft schon überprüfen, ob die zugewiesenen Speicherbereiche eingehalten werden, bei anderen Sprachen erlauben moderne Compiler (oder Compilererweiterungen) die Erzeugung von Überprüfungscode.

(5) Welchen Wert auf dem Stack möchte der Angreifer manipulieren und warum?

Die Rücksprung Adresse, damit nach Ausführen der Funktion an eine neue nicht programmeigene Adresse gesprungen wird. Zum Beispiel könnte zurück in den Buffer gesprungen werden, um dort gespeicherten Code auszuführen.

(6) Was möchte der Angreifer erreichen und warum ist es lohnenswert?

Das Ziel ist es, den Zugang zu einem System zu erlangen, etwa durch öffnen einer Shell, damit problemlos weiterer Code ausgeführt werden kann.

Bonusfragen:

(g) Warum muss der Shellcode in Assemblersprache übersetzt werden?

Der Rest des Programms wurde ja bereits bei der Kompilierung in maschinenausführbaren Code übersetzt. Wenn jetzt eine Rücksprungadresse manipuliert wurde, muss an der neuen Stelle auch direkt von der Maschine ausführbarer Code liegen, also Assembler. 

(f) Fassen Sie den Angriff Buffer overflow zusammen.

Der Buffer overflow (Pufferüberlauf) Angriff ist eine der verbreitetsten Angriffsmethoden. Es werden dabei Fehler in Programmen ausgenutzt, die es erlauben zu große Datenmengen in zu kleine Speicherbereiche (den Buffer) zu schreiben. Wenn der reservierte Speicher voll ist, werden die nachfolgenden Speicherbereiche auch mit überschrieben, der Speicher läuft also über. So können Daten in Speicherbereiche geschrieben werden, die eigentlich nicht zugänglich sein sollten. Mit dem Angriff kann ein instabiler Programmablauf provoziert, Daten verfälscht und im schlimmsten Fall schadhafter Programmcode ausgeführt werden. Die Ausführung von schadhaftem Programmcode gelingt in der Regel durch Manipulation von Sprungadressen.

(g) Wie kann ein Betriebssystem einen Buffer overflow verhindern?

Es ist wichtig, dass die für das Betriebssystem spezifischen Standard-Bibliotheken der verwendeten Programmiersprache sorgfältig programmiert sind.

Das Betriebssystem kann mit Executable Space Protection (ESP) verhindern, dass Programmcode in nicht zugelassenen Speicherbereichen ausgeführt wird.

112

Aufgabe 2. Hier sind nur die von uns in Trashcan\_Framework.c modifizierten Funktionen. Die anderen (main, parse) sind die vorgegeben.

```
#define BUFSIZE          1024
#define PATHSIZE         256
#define TRASHFOLDER      ".ti3_trash"

char copy_buffer[BUFSIZE];

int copy(char *source, char *target)
{
    ssize_t readen;
    ssize_t written;
    int h_target;
    int h_source;
```

```
if ( h_source = open(source, O_RDONLY), h_source == -1 )
    return -1;
if ( h_target = open(target, O_WRONLY|O_CREAT|O_EXCL, 0644), h_target == -1 )
    return -2;
while ( readen = read(h_source, copy_buffer, BUFSIZE), readen > 0 )
{
    written = write(h_target, copy_buffer, readen);
    if ( readen-written != 0 )
        return -3;
}
close(h_source);
close(h_target);
return 0;
}

char* getpath(char *dirname, char *filename)
{
    ssize_t length = strlen(dirname)+strlen(filename)+2;
    char *path = (char*) malloc(length);
    strcpy(path, dirname);
    strcat(path, "/");
    strcat(path, filename);
    return path;
}

int setup_trashcan(char *dirname){
    return mkdir(dirname, 0755);
}
int put_file(char *dirname, char *filename)
{
    char *source = filename;
    char *target = getpath(dirname, filename);
    int err;
    if ( err = copy(source, target), err < 0 )
        return err;
    if ( remove(source) == -1 )
        return -3;
    return 0;
}
int get_file(char *dirname, char *filename)
{
    char *source = getpath(dirname, filename);
    char *target = filename;
    int err;
    if ( err = copy(source, target), err < 0 )
        return err;
    if ( remove(source) == -1 )
        return -3;
    return 0;
}
int remove_file(char *dirname, char *filename)
{
    char *path = getpath(dirname, filename);
    if ( remove(path) == -1 )
        return -1;
    return 0;
}
```