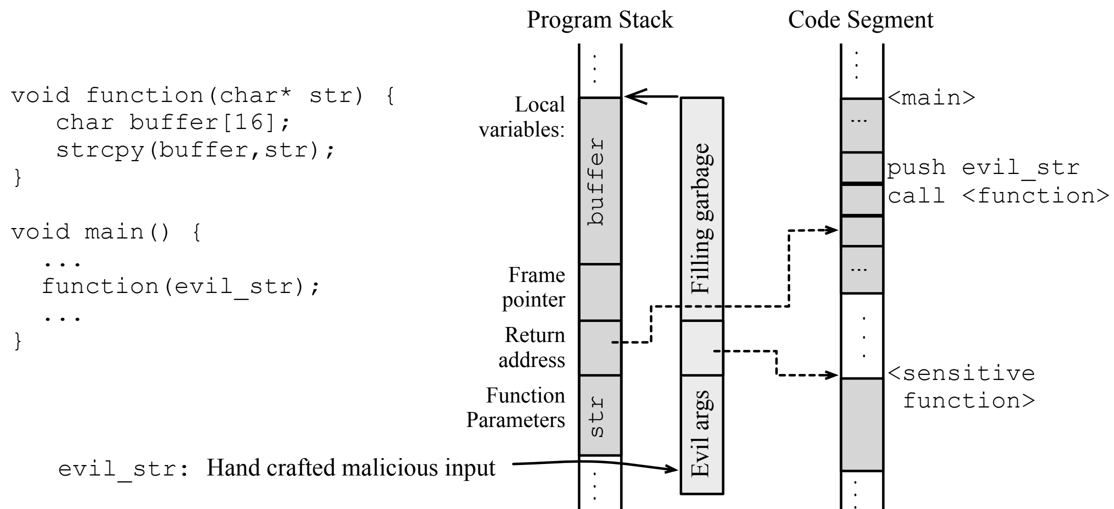
DCC888 – Tainted Flow Analysis

Name: ______________________________ ID: ______________

1. A *buffer overflow* consists in writing in a buffer a quantity of data large enough to go past the buffer's upper bound; hence, overwriting other program or user data. It can happen in the stack or in the heap. In the *stack overflow* scenario, by carefully crafting this input string, one can overwrite the return address in a function's activation record; thus, diverting execution to another code area. The first buffer overflow attacks included the code that should be executed in the input array. However, modern operating systems mark writable memory addresses as non-executable – a protection mechanism known as *Read*⊕*Write*. Therefore, attackers tend to divert execution to operating system functions such as `chmod` or `sh`, if possible. Usually the malicious string contains also the arguments that the cracker wants to pass to the sensitive function. Below we illustrates an example of buffer overflow.



A buffer overflow is a kind of tainted flow attack, as we have malicious information flowing into sensitive functions. In this question you must explain how this kind of attack can fit into the tainted flow framework.

(a) Explain which operations or functions are the *sources*, *sanitizers* and sinks that we are likely to find in a typical C program.

(b) Consider the toy language seen in Figure 1. You shall use this language to explain how we can implement a tainted flow analysis that tracks buffer overflow

Programs (P)	::=	$\ell_1 : I_1, \ell_2 : I_2, \ldots, \ell_n : \texttt{end}$
Labels (L)	::=	$\{\ell_1, \ell_2, \ldots\}$
Variables (V)	::=	$\{v_1, v_2, \ldots\}$
Constants (C)	::=	$\{c_1, c_2, \ldots\}$
Operands (O)	::=	$V \cup C$
Instructions (I)	::=	
– Assignment	|	$v = o$
– Input	|	$v = \bullet$
– Binary operation	|	$v_1 = v_2 \oplus v_3$
– ϕ-function	|	$\overline{v} = \phi(\overline{v_1}, \ldots, \overline{v_n})$
– Store into memory	|	$*v_1 = v_3$
– Load from memory	|	$v_1 = *v_2$
– Branch if zero	|	$\texttt{brz}(v, \ell)$
– Unconditional jump	|	$\texttt{jmp}(\ell)$

Figure 1: The syntax of the toy language that we shall use to explain how the tainted flow framework could be used to track buffer overflow vulnerabilities.

vulnerabilities. In this question, you must design transfer functions for each of these instructions. Assume that every program implemented in our toy language must be in the static single assignment format.

(c) Figure 2 shows a program. This program contains a buffer overflow vulnerability: if the adversary assigns to `N` a value that is high enough, he or she will be able to read the contents of memory which are, possibly, secret information. The rest of this question refers to this program.

i. Is there a chain of dependences between `N` and the load at label ℓ_7?

ii. Does your analysis catches the buffer overflow vulnerability in the program of Figure 2?

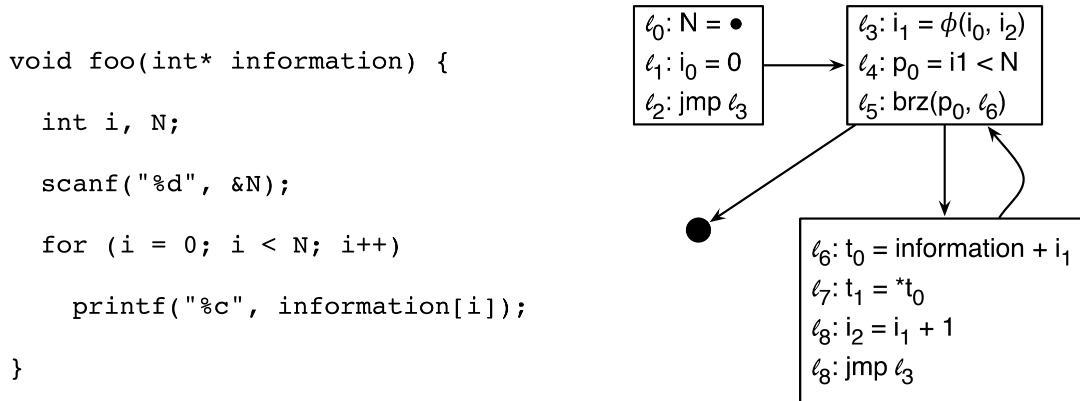


Figure 2: Program that contains a buffer overflow vulnerability.

2. Lets try to give to our tainted flow analysis a bit more of notation. In this exercise, you must model the data-flow equations used in our tiny, nano-PHP language, whose syntax is given below:

Name	**Instruction**	**Example**
Assignment from source	$x = \circ$	`$a =$_POST['content']`
Assignment to sink	$\bullet = v$	`echo($v)`
Simple assignment	$x = \otimes(x_1, \ldots, x_n)$	`$a = $t1 * $t2`
Branch	`bra` $l_1, \ldots, l_n$	general control flow
Filter	$x_1 =$ `filter`	`$a = htmlentities($t1)`
Validator	`validate` x, l_c, l_t	`if (!is_numeric($1)) abort();`

A label $l \in L$ refers to a program location and is associated to one instruction. A nano-PHP program is a sequence of labels, $l_1 l_2 \ldots l_{exit}$. We use the symbol $\otimes$ to denote any operation that uses a sequence of variables to define another variable. The symbol *Var* denotes the domain of program variables. The symbol *Abs* denotes the domain of abstract states $\{\bot, \text{clean}, \text{tainted}\}$. We define a lattice $(Abs, <)$ by augmenting the set *Abs* with the following ordering $\bot < \text{clean} < \text{tainted}$. We represent data-flow information with the function $[\![_]\!] : (L \times L) \rightarrow Var \rightarrow Abs$. This function associates to each program point (l, l') a map storing the abstract values of each program variable. We use the notation $[\![l_1, l_2]\!]$ to denote information at (l_1, l_2).

In this exercise you must define the transfer functions $(Var \rightarrow Abs) \rightarrow (Var \rightarrow Abs)$ that are associated with each instruction. The initial state of the analysis associates undefined with all program variables at every point. If we let $PRED(l)$ be the set of program points immediately before label l, then we define the auxiliary function *JOIN*

as $JOIN(l) = \bigsqcup [\![l_i, l]\!], l_i \in PRED(l)$. The goal of the exercise is to fill up the rest of the table below. We have started this process, showing the transfer function associated with $x = \circ$. Notice that we denote the successor of a given label l by l_+, whenever this successor is unique. Also, notice that we use the notation $f \setminus [x \mapsto y]$ to denote the updating of function f. In other words, if $g = f \setminus [x \mapsto y]$, then $g(x) = y$, and $g(x') = f(x')$ whenever $x \neq x'$.

l	$[\![\text{-}]\!]$
$x = \circ$	$[\![l, l_+]\!] = JOIN(l) \setminus [x \mapsto \text{tainted}]$
$\bullet = x$	
$x = \otimes(x_1, \ldots, x_n)$	
`bra` $l_1, \ldots, l_n$	
$x =$ `filter`	
`validate` x, l_c, l_t	

3. A side-channel is a flaw in the implementation of a cryptosystem that gives an adversary the opportunity to learn information considered secret. Timing information is one of the most exploited kinds of side-channels. The time behavior of a cryptosystem is very dependent on its implementation; hence, vulnerabilities related to time-based leaks are often overlooked during the design of a cryptographic algorithm.

   (a) Observe the program in Figure 3. This program contains a time-based side channel vulnerability. In such vulnerability, the adversary monitors small fluctuations in the execution time of a cryptographic algorithm. These time variations are due to conditional branching, variable-time instruction-level optimization, memory hierarchy performance or communication latency. In this question we shall focus only on control flow: we want to know if secret information can control the runtime of a program. Explain how an adversary can use the side-channel in Figure 3 to recover the seed of the random number generator that function `genKeyMask` uses.

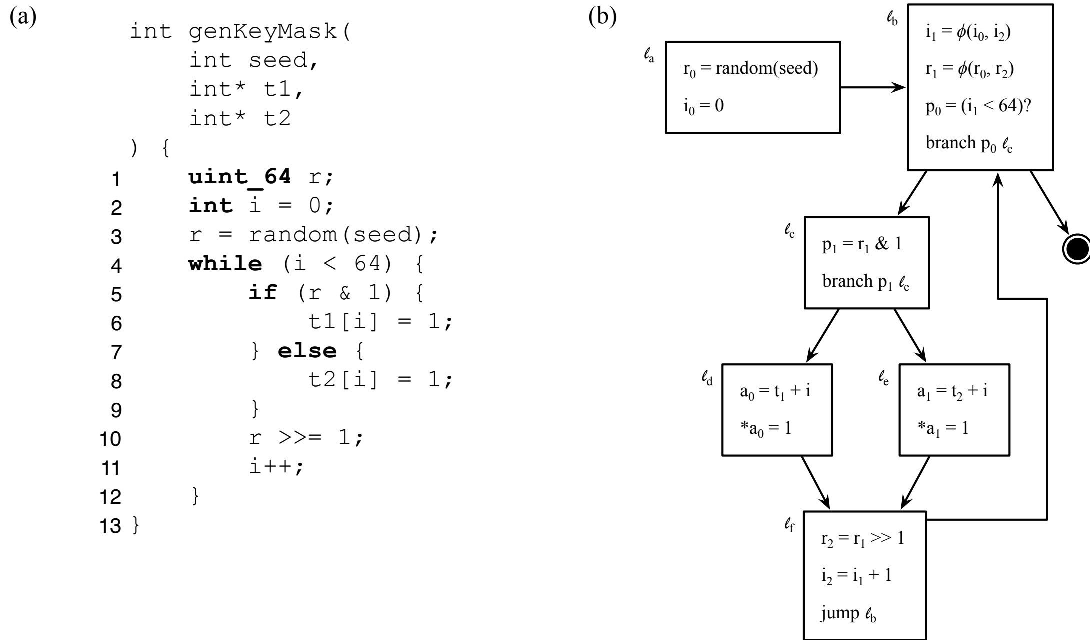


Figure 3: (a) Program with sensitive information, e.g., argument "seed". (b) Control flow graph of the program in Static Single Assignment form.

(b) Discuss a solution, in the tainted flow framework, to detect time-based side channel vulnerabilities in programs.

i. What are the sinks?

ii. What are the sources?

iii. Is there any kind of validator?

iv. How can we use the tainted flow framework to uncover side-channels in programs?