FORMAL REASONING OF VARIOUS CATEGORIES OF WIDELY EXPLOITED SECURITY VULNERABILITIES USING POINTER TAINTEDNESS SEMANTICS[1]

Shuo Chen, Karthik Pattabiraman, Zbigniew Kalbarczyk and Ravi K. Iyer
Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1308 W. Main Street, Urbana, IL 61801. {shuochen , pattabir, kalbar, iyer}@crhc.uiuc.edu

Abstract: This paper is motivated by a low level analysis of various categories of severe security vulnerabilities, which indicates that a common characteristic of many classes of vulnerabilities is pointer taintedness. A pointer is said to be tainted if a user input can directly or indirectly be used as a pointer value. In order to reason about pointer taintedness, a memory model is needed. The main contribution of this paper is the formal definition of a memory model using equational logic, which is used to reason about pointer taintedness. The reasoning is applied to several library functions to extract security preconditions, which must be satisfied to eliminate the possibility of pointer taintedness. The results show that pointer taintedness analysis can expose different classes of security vulnerabilities, such as format string, heap corruption and buffer overflow vulnerabilities, leading us to believe that pointer taintedness provides a unifying perspective for reasoning about security vulnerabilities.

Key words: Security Vulnerability, Static Analysis, Program Semantics, Equational Logic, Pointer Taintedness

[1] This work is supported in part by a grant from Motorola Inc. as part of Motorola Center for Communications, in part by MURI Grant N00014-01-1-0576, and in part by NSF CCR 00-86096 ITR.

1. INTRODUCTION

Programming flaws that result in security vulnerabilities are constantly discovered and exploited in real applications. A significant number of vulnerabilities are caused by improper use of library functions in programs. For example, omitting buffer size checking before calling string manipulation functions, such as *strcpy* and *strcat*, causes many buffer overflow vulnerabilities. Passing user input string as the format string in *printf*-like functions causes format string vulnerabilities. Heap corruption vulnerabilities are the result of invoking the *free* function with a pointer pointing to an overflowed buffer or a buffer that has not been allocated by the heap manager. Library functions are usually secure only under certain conditions, and therefore, formally extracting security conditions from library code can be a valuable aid in implementing applications free of security vulnerabilities.

We introduce the notion of pointer taintedness as a basis for reasoning about security vulnerabilities. The notion of pointer taintedness is based on the observation that the root cause of many reported vulnerabilities is due to the fact that a pointer value (including return address) can be derived directly or indirectly from user input. Since pointers are internal to applications, their values should be transparent to users. Thus a taintable pointer is a potential security vulnerability. By analyzing the application source code, the potential for pointers to be tainted can be determined and hence possible vulnerabilities can be identified.

Existing compiler-based techniques, such as CQUAL [7] and SPLINT [1], perform taintedness analysis by associating an attribute or a type qualifier with program symbols, i.e., variables, constants, arguments and return values. Although these techniques allow reasoning about taintedness of symbols, they cannot analyze the taintedness of pointers unless C statements explicitly perform the tainting. As we will show, there are many situations (e.g., format string vulnerabilities, heap corruptions and stack buffer overflows) where pointers become tainted without explicit assignment statements in the code. Pointer taintedness that leads to well known vulnerabilities usually occurs as a consequence of low level memory writes, typically hidden from the high level code. Hence a memory model is necessary to reason about pointer taintedness. This paper proposes a formalization to define and analyze pointer taintedness so as to uncover potential security vulnerabilities. We focus on commonly used library functions because many widely exploited vulnerabilities are caused by pointer taintedness occurring in library functions.

A memory model is formally defined in this paper using equational logic[2], which forms the basis of the semantics of pointer taintedness. A mechanical reasoning technique is applied to several library functions to examine the possibility of a pointer being tainted. The analysis process for each library function yields a set of formally specified preconditions that must be satisfied to eliminate the possibility of pointer taintedness. These pre-conditions either correspond to already known vulnerability scenarios (e.g., format string vulnerability and heap corruption) or indicate the possibility of function invocation scenarios that may expose new vulnerabilities.

2. RELATED WORK

Security vulnerabilities have been reported in many applications. The *Bugtraq* vulnerability list and *CERT* advisories maintain information about reported vulnerabilities. Security vulnerabilities can be modeled using finite state machines, by breaking them into multiple simple operations, each of which may be associated with one or more logical predicates [10].

Many static detection techniques have been developed based on the recognition of existing categories of security vulnerabilities. Techniques such as [1] and [2] can check security properties if vulnerability analysts are able to specify them as annotations in the code. Domain-specific techniques require less human effort, but each technique only detects a specific type of vulnerability. Static detection techniques are proposed to detect to buffer overflow vulnerabilities, e.g. [6]. Runtime mechanisms against security attacks have been introduced in [8] and [9]. Xu et. al. [5] recently proposed efficient approaches to randomize memory layout and to encode control flow information, which aims at defeating various attacks in a generic manner.

The notion of taintedness was first proposed in Perl language as a security feature. Inspired by this, static detection tools like SPLINT [1] and CQUAL [7] also use taintedness analysis to guarantee that user input data is never used as the format string argument in *printf*-like functions. In both these tools, taintedness is an attribute associated with C program symbols. A symbol gets tainted only if an explicit C statement passes a tainted value to it by assignment, argument passing or function return. As shown in the next section, in many real attacks, pointers are tainted without explicit C statements tainting program symbols. Since these tools do not have a memory model, they cannot determine whether an address is tainted and

[2] An introduction to equational logic can be found in [11].

hence cannot reason about the underlying memory status. In [12], several examples are provided where data can be tainted without being detected by SPLINT and CQUAL. The inability to reason about taintedness at the memory level is an inherent limitation of existing taintedness analysis tools.

3. POINTER TAINTEDNESS EXAMPLES

Our study indicates that many known security vulnerabilities, such as format string, heap corruption, buffer overflow and *Glibc glob* vulnerabilities, arise due to pointer taintedness. In these examples, the pointers become tainted without explicit C assignment statements. Due to space limitations, only the example of the format string vulnerability is discussed in detail. Other examples are described in the extended version of this paper [12]. In this section, we first give a high level description (Table 1) of the format string vulnerability, and show how it can be exploited. We then show that a memory model (Figure 1) is necessary for reasoning about pointer taintedness.

Table 1: Format String Vulnerability Illustration

int Vfprintf (FILE *s, const char *format, va_list ap)
{ char * p;
int count;
p = format; … …
L1: *(int *) ap = count;
… …
}
 	
int Printf (const char *format, ...)
{ va_list arg; … …
L2: Vfprintf (stdout, format, arg);
… …
}
 	
int i,j;
int main()
{ char buf[100];
	//This is how to call *Printf* correctly
	strcpy(buf,"hello");
	i=1234;
	L3: Printf("string=%s\ndata=%d\n%n",buf,i,&j);
	Printf("total output length=%d\n",j);
	
	//This is how format string vulnerability occurs
	scanf("%s",buf);
	L4: Printf(buf);
	L5: Printf("\ni=%d\n",i);
	}
	
	Program Output:
	O1: string=hello
	O2: data=1234
	O3: total output length=23
	O4: 134514747123413451916812345
	O5: i=31

The format string vulnerability is caused by incorrect invocation of *printf*-like functions (e.g., *printf, sprintf, snprintf, fprintf* and *syslog*). Table 1 gives examples of correct and incorrect invocations of *Printf()* (a simplified version of LibC *printf()* that we developed). This sample program produces five lines of output (shown in Table 1). Output lines O1 and O2 result from executing line L3 of the code: the string *buf* hosting "*hello*", and the integer *i* has the value 1234. In addition, line L3 uses the format directive *%n* to write the character count (i.e., the number of characters printed up to that point) into the address of the corresponding integer variable. In this case, the length of "*string=hello¶data=1234¶*" is 23, so *Printf()* writes 23 to the integer *j*.

This value is printed out in line O3 of the program output. The format string vulnerability is caused by incorrect invocation of *Printf* in line L4, which directly uses *buf* as the format string (the proper usage should be *Printf("%s", buf)*).

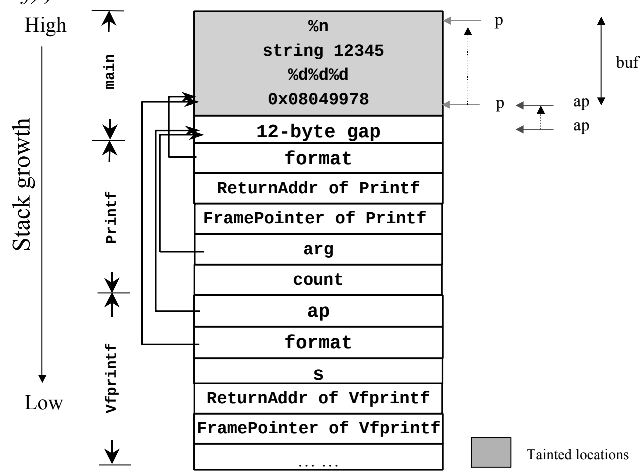


Figure 1: How to Overwrite the Global Integer *i*

We now show how an attacker can exploit this vulnerability. Let's assume that the attacker wants to corrupt an arbitrary memory location (e.g., the global integer *i*). In order to do this, he/she constructs an input string *buf* as given below (observe that the beginning of the input string corresponds to the address of global integer *i*):

\x78 \x99 \x04 \x08 %d %d %d '1' '2' '3' '4' '5' %n

The string is read by *scanf()* and passed to *Printf()*, which in turn, calls *Vfprintf()*. Just before Line L1 is executed, the stack layout is like the one in Figure 1. In *Vfprintf()*, there are two pointers: *p* is the pointer to sweep over the format string *buf* (from "\x78" to"%n"), and *ap* is the pointer to sweep over the arguments (starting from the 12-byte gap). The attacker deliberately embeds three "%d" directives in *buf* so that *ap* can consume the 12-byte gap and get to the word 0x08049978. A padding string "12345" follows the "%d" directives in order to adjust the character count. As we see in the program output line O4, the words in the 12-byte gap are printed as three integers followed by a padding string "12345". Eventually, when *p* arrives at the position of "%n" (i.e., the code line L1 is about to be executed), *ap* happens to arrive at the position of 0x08049978. Line L1 writes the character count *count* to the location pointed by **ap*. In this case, since the content in the location pointed by **ap* is the address of the integer *i*, the character count 31 is written to *i*. Note that this attack can overwrite any memory location, including locations containing return addresses or the global offset table of an application, which can result in the execution of the attacker's code.

We can view the above vulnerability as a consequence of pointer taintedness. In the above code (Table 1), the string *buf* is obtained from user input and is hence tainted (as indicated in Figure 1 as a grey area). When the pointer *ap* sweeps over the stack and points to *buf*, **ap* becomes tainted. *ap* is then dereferenced in Line L1, and the tainted value of **ap* is the target address of the write operation. This can lead to the corruption of an arbitrary memory location. Thus we see that pointer taintedness is the root cause of this vulnerability. Note that the pointer **ap* gets tainted because *ap* moves into the tainted memory locations, and there is no explicit assignment of a tainted value to **ap* in the C code. Hence a memory model is necessary to reason about the taintedness of **ap*. The next section defines the formal semantics of pointer taintedness using a memory model, and Section 5 shows how the semantics can be used to reason about security vulnerabilities in library functions.

4. SEMANTICS FOR POINTER TAINTEDNESS

Starting with the programming semantics of Goguen and Malcolm [3], this section proposes a formal semantics to reason about pointer taintedness in programs. The semantics proposed in [3] defines instructions, variables and expressions. We extend this semantics to include memory locations and addresses. Using the memory model, the notion of taintedness is incorporated into the semantics.

We define tainted data as: (1) data coming from input devices (e.g., by *scanf(), fscanf(), recv(), recvfrom()*), or (2) data copied or arithmetically calculated from tainted data. A tainted pointer is a pointer whose value (semantically equivalent to “data”) is tainted. This definition can be formalized in equational logic using the Maude tool [4], which we used to reason about pointer taintedness.

In the semantics defined in [3], a *Store* represents the current state of all program variables. We extend this definition of a *Store* to be a snapshot of the entire memory state at a point in the program execution. The execution of a program instruction is defined as a function taking two arguments, a *Store* and an instruction, and producing another *Store*. There are two attributes associated with every memory location of a *Store*: *content* and *taintedness*. Accordingly, two operations, *fetch* and *location-taintedness*, are formally defined. The *fetch* operation `Ftch(S,I)` gives the content of the address `I` in store `S`; the *location-taintedness* operation `LocT(S,I)` returns a Boolean value indicating whether the content of the specified address is tainted.

There is no notion of "variable" in this semantics. Any variable in a C program is mapped to a memory location addressed by the integer with the same name as the program variable. For example, the C program variable foo is mapped as a memory location addressed by the integer foo. We define the ^ operator to dereference an integer, i.e., to fetch the location addressed by the integer. Note that the address (a.k.a., the left value) of the C program variable foo is represented by the integer foo in the semantics; and the content (a.k.a., the right value) of the C program variable foo is represented by (^ foo). The expressions in the semantics are arithmetic operators (e.g., +, - and *) concatenating integers and integer dereferences. For example, expression 200+(^foo) represents "200 plus the content of the C program variable foo". Expression 200+foo represents "200 plus the address of the C program variable foo".

We define two operations – *evaluation* and *expression-taintedness* – for expressions, based on the *fetch* and *location-taintedness* operations. The *evaluation* operation Eval(S,E) gives the result of evaluating the expression E under store S; the *expression-taintedness* operation ExpT(S,E) indicates whether expression E contains any data from a tainted location, e.g., ExpT(S,(^foo)+2) indicates whether the expression (^foo)+2 contains any data from a tainted location, which is equivalent to checking whether the memory location addressed by foo is tainted. *Thus pointer taintedness is defined as a dereference of a tainted expression.*

Table 2 lists a set of axioms for the *evaluation* and *expression-taintedness* operations, and gives examples of applying the equations.

Table 2: Axioms of *Evaluation* and *Expression-Taintedness* Operations

	Axioms	**Examples**
1	Eval(S,I) = I	Eval(S,5) = 5
2	Eval(S, ^ E1) = Ftch(S, Eval(S,E1))	Eval(S, ^ foo) = Ftch(S,Eval(S,foo)) = Ftch(S,foo)
3	Eval(S, - E1) = - Eval(S, E1)	Eval(S, - 30) = - Eval(S, 30) = - 30
4	Eval(S, E1 – E2) = Eval(S, E1) - Eval(S, E2)	Eval(S, 3-2) = Eval(S, 3) – Eval(S, 2) = 3–2 = 1
5	Eval(S, E1 + E2) = Eval(S, E1) + Eval(S, E2)	Eval(S, 3+2)=Eval(S, 3)+Eval(S,2) = 5
6	Eval(S, E1 * E2) = Eval(S, E1) * Eval(S, E2)	Eval(S, 3*2)=Eval(S, 3)*Eval(S, 2) = 6
7	ExpT (S,I) = false	ExpT(S,5) = false
8	ExpT(S, ^ E1) = LocT(S,Eval(S,E1))	ExpT(S,^foo) =LocT(S,Eval(S,foo)) =LocT(S,foo)
9	ExpT(S, - E1) = ExpT(S,E1)	ExpT(S, -5) = ExpT(S, 5)= false
A	ExpT(S,E1-E2)=ExpT(S,E1) or ExpT(S,E2)	ExpT(S, (^foo)–2) = ExpT(S, (^foo)) or ExpT(S, 2) = LocT(S,foo) or false = LocT(S,foo)
B	ExpT(S,E1+ E2)=ExpT(S,E1)or ExpT(S,E2)	ExpT(S, (^foo)+2) = … = LocT(S,foo)
C	ExpT(S,E1*E2)=ExpT(S,E1) or ExpT(S,E2)	ExpT(S, (^foo)*2) = … = LocT(S,foo)

Lines 1-6 define how to evaluate an expression under store S. For example, line 1 indicates that the evaluation result of a constant I under store

S is the constant I. Line 2 indicates that Eval(S,^E1) can be computed by first evaluating E1 under S, then applying *fetch* operation on the evaluation result. The semantics of arithmetic operations are defined in Lines 3-6.

Lines 7-C define the *expression-taintedness* operator. Note that the relationship between the *expression-taintedness* and *location-taintedness* operators is similar to the relationship between the *evaluation* and *fetch* operators. Line 7 indicates that an integer constant is not a tainted expression. Line 8 indicates that determining whether the expression ^E1 is tainted is equivalent to checking whether the location addressed by the evaluation result of E1 is tainted. Line B indicates that the expression E1+E2 is tainted if either E1 or E2 is tainted. The example of Line C shows that the expression (^foo) + 2 is tainted if and only if the location pointed to by foo is tainted, according to the equation in Lines 7 and 8.

Table 3: Semantics of Statements

Statement	Semantics
mov [E1] <- E2	Move the evaluation result of the expression E2 to the memory location addressed by the evaluation result of the expression E1
if T then P1 else P2 fi	If the condition T is true, Execute P1 otherwise execute P2
while T do P od	If the condition T is true, execute P, repeat until T is false

Table 3 gives the informal semantics of a subset of the supported statements. Their formal semantics are similar to the specifications given in [3], and are sufficient to analyze a wide variety of program constructs in the C language. However, they are not sufficient to faithfully model all C statements. For example, the program counter has not been defined in the semantics. So certain C statements, such as goto, break, continue, return and exit cannot be modeled, but it is relatively easy to extend the semantics for these also.

Formal specifications of statements other than the mov statement are fairly straightforward. Axioms defining mov statement semantics are shown in Table 4. The goal is to define the *fetch* (Ftch) and *location-taintedness* (LocT) operations after applying a mov instruction on store S.

Table 4: Equations Defining mov Statement Semantics

1	Ftch((S ; mov [E1] <- E2),X1) = Eval(S,E2) if (Eval(S,E1) is X1) .
2	Ftch((S ; mov [E1] <- E2),X1) = Ftch(S,X1) if not (Eval(S,E1) is X1) .
3	LocT((S ; mov [E1] <- E2),X1) = ExpT(S,E2) if (Eval(S,E1) is X1) .
4	LocT((S ; mov [E1] <- E2),X1) = LocT(S,X1) if not (Eval(S,E1) is X1) .

The semicolon operator in our notation represents the execution of an instruction on a store, which results in a new store. For example, (S;mov[E1]<-E2) is the store after executing mov[E1]<-E2 on store S. Line 1 indicates that if the expression E1 evaluates to X1 under store S, then when fetching the location X1 after executing the instruction mov[E1]<-E2, we get the evaluation result of E2 under store S. Line 2 indicates that if the expression E1 does not evaluate to X1 under S, then when fetching the location X1 after executing the instruction mov[E1]<-E2, we still get the content in the location X1 under S (i.e., before executing the instruction mov[E1]<-E2). Similarly, the LocT operation is defined for mov statement in Lines 3 and 4.

5. FORMAL REASONING ON POINTER TAINTEDNESS VIOLATIONS

This section presents pointer taintedness analysis for three common library functions based on the defined semantics, and extracts their associated security preconditions. The analysis identifies several known vulnerabilities, such as format string, buffer overflow and heap corruption vulnerabilities, thereby showing that pointer taintedness based reasoning is able to unify different kinds of vulnerabilities.

Our experience suggests that statements needing critical examination for pointer taintedness are typically indirect writes, where a pointer points to a target address to be written, e.g., the pointer p in *p = foo and memcpy(p,foo,10). Checking indirect write statements is important because these statements can result in two types of pointer taintedness violations. For example, in the statement *p = foo, (1) if the value of p is tainted, then data foo can be written to any memory location; (2) if p is a pointer to a buffer but points to a location outside the buffer, then the statement *p = foo can taint the memory location p points to, which may be a location of a return address, a function frame pointer or another pointer.

5.1 Analysis of *strcpy()*

A simple but interesting example is *strcpy(),* which copies a NULL-terminated source string to a destination buffer. The string manipulation functions, including *strcpy()*, *strcat()* and *sprintf()*, are known to cause a significant number of buffer overflow vulnerabilities. Our formal reasoning extracts security preconditions from the implementation of *strcpy()*. The source code of *strcpy()* and its formal representation are given in Table 5.

As the only indirect write operations in the source code are in Line 1 and Line 2, it is enough to prove the three theorems listed in Table 6. We assume that the NULL-terminator (i.e., the character '\0') of the source string src is at the location (src + srclen), and that the size of the buffer dst is dstsize. Theorem NV1 ensures that before Line L1, the content of the variable dst is not tainted; Theorem NV2 ensures that for any address A outside the buffer dst (from dst to dst+dstsize), the taintedness of location A after Line L1 is same as the taintedness of location A before Line L0. NV3 is similar to NV1, but proves the property for the memory state before Line L2 is executed.

Table 5: Source Code and Formal Semantics of *strcpy()*

`char * strcpy (char * dst, char * src) {`
`        char * res;`
`0:      res =dst;`
`        while (*src!=0) {`
`1:        *dst=*src;`
`          dst++;`
`          src++;`
`         }`
`2:      *dst=0;`
`        return res; }`
	
	`L0:  mov [res] <- (^ dst) ;`
	`     while (~((^ ^ src) is 0)) do`
	`L1:     mov [^ dst] <- (^ ^ src) ;`
	`        mov [dst] <- (^dst + 1) ;`
	`        mov [src] <- (^ src + 1)`
	`     od ;`
	`L2:  mov [^ dst] <- 0 .`

Table 6: Theorems to Prove for Function *strcpy()*

Theorem NV1: If S1 is the store before Line L1, then LocT(S1,dst) = false
Theorem NV2: If S0 is the store before Line L0, and S2 is the store after Line L1, $\forall$ A $\in$ INT $\bullet$ A < Eval(S0, ^dst) or Eval(S0, ^dst+dstsize) $\le$ A => LocT(S2,A) = LocT(S0, A)
Theorem NV3: If S3 is the store before Line L2, then LocT(S3,dst) = false

Table 7 gives a set of security preconditions extracted in the process of proving the theorems. Among the four preconditions, Condition 4 is known because of the large number of buffer overflow vulnerabilities caused by string manipulation. This condition has already been documented on Linux MAN page of *strcpy*. Condition 2 indicates the scenario of overlap between src and dst and is examined further in Section 6.2. Violation of Condition 3 may occur when a program miscalculates the location of a stack buffer, causing the function frame of *strcpy()* to be covered by the buffer and is discussed in Section 6.1.

Table 7: Sufficient Conditions to Ensure the Validity of Theorems NV1 – NV3

1. Initially, the location of dst is not tainted.
2. The buffers src and dst do not overlap in such a way that the buffer dst covers the NULL-terminator of the src string.
3. The buffer dst does not cover the function frame of *strcpy()*, which consists of the locations &dst, &src and &res.
4. srclen < dstsize

5.2 Analysis of *Free()*

We implemented a binary buddy heap management system including function *Malloc()* and *Free()*. The memory block to be freed is pointed to by pointer FreedBlock. The binary buddy heap management algorithm requires the deallocated memory block FreedBlock to be merged with its buddy block if the buddy block is also free. The pointer BuddyBlock points to the buddy block. FreedBlock and BuddyBlock are structs of type HEAP_BLOCK as shown in Table 8. The Size field indicates the size of the memory chunk. The busy field indicates whether the memory chunk is free. Fields Fwd and Bak are pointers to maintain a doubly-link list of free memory chunks.

Table 8: Indirect Write Statements in *Free()* Source Code

```
typedef struct _HEAP_BLOCK {
   int          Size;         // The size of the block.
   int          Busy;         // Is this block busy?
   struct _HEAP_BLOCK * Fwd,* Bak; // List to the free blocks of the same size
} HEAP_BLOCK;
```

There are three lines in the *Free()* function where indirect write operations are performed. Six pointers are involved in the operations, including FreedBlock, BuddyBlock, FreedBlock->Fwd, FreedBlock->Bak, BuddyBlock->Fwd and BuddyBlock->Bak. Table 9 states the theorem to be proved for conditions guaranteeing that pointers are not tainted. It is assumed that the offset of the Fwd field in the HEAP_BLOCK structure is 2, and that the offset of the Bak field is 3. Theorem NV1 ensures that none of the six pointers is tainted before executing any indirect writes.

Table 9: Theorems to Prove for Function *Free()*

Theorem NV1: If S is the store before executing the indirect writes, then	
ExpT(S, (^ FreedBlock)) = false and	//FreedBlock is not tainted
ExpT(S, (^BuddyBlock)) = false and	//BuddyBlock is not tainted
ExpT(S, ^((^ FreedBlock) + 2)) = false and	//FreedBlock->Fwd is not tainted
ExpT(S, ^((^ FreedBlock) + 3)) = false and	//FreedBlock->Bak is not tainted
ExpT(S, ^((^ BuddyBlock) + 2)) = false and	//BuddyBlock->Fwd is not tainted
ExpT(S, ^((^ BuddyBlock) + 3)) = false	//BuddyBlock->Bak is not tainted

The process of proving the theorems extracted a set of formally specified conditions that guarantee the validity of Theorem NV1. Table 10 describes the conditions. The function *Free()* is safe to be called when the caller function can guarantee these conditions. Violations of condition 1 are unlikely to occur, and the same is true for condition 7. Violations of condition 6 cause the classic double-free errors, and violations of condition 3 and 4 lead to the popular heap buffer overflow vulnerability. An example illustrating violation of condition 2 is presented in Section 6.3.

Table 10: Sufficient Conditions to Ensure the Validity of Theorem NV1

1. The memory range of the heap and the memory range of the current function frame do not overlap.
2. Immediately before *Free()* is called, FreedBlock points to a location on the heap.
3. Immediately before *Free()* is called, the Fwd and Bak links of the block of FreedBlock are not tainted.
4. All free-chunk double-linked lists are within the heap range, i.e., no Fwd or Bak links points to any location outside the heap.
5. No Fwd or Bak pointers in any free-chunk double-linked list are tainted.
6. Before *Free()* is called, FreedBlock is not linked in any free-chunk list.
7. If BuddyBlock is free, then BuddyBlock is linked in a free-chunk double-linked list.

5.3 Analysis of *Printf()*

We implemented a function *Printf()*, similar to the LibC function *printf()*, except that *Printf()* calls its child function *Vfprintf()*, which is a simplified version of LibC function *vfprintf()*. *Vfprintf()* implements the format directives *%%*, *%d*, *%s* and *%n*. The total length of *Vfprintf()* is 55 lines. Pointer p is used to sweep over the format string format. The argument list is swept over by pointer ap.

There are only two lines of indirect write operations in the function (Table 11). Line L1 is to get the last digit of data and save it in the nth position of the buffer buf. Line L2 is to assign the character count to the memory location pointed by the current argument. Note that ap is the argument list pointer pointing to the current argument.

Corresponding to the two indirect write operations, we need to prove the theorems given in Table 12. Theorem NV1A ensures that before executing code in Line L1, the memory location containing the variable n is not tainted. Theorem NV1B ensures that after Line L1, the memory location buf+10 is not tainted. Theorem NV1A and NV1B can be easily proved by the theorem prover. Theorem NV2 ensures that before Line L2, the expression (^ ^ ap) is not tainted, i.e., the memory location pointed by (^ ap) is not tainted, i.e., the memory location pointed by the content of variable ap is not tainted. The preconditions extracted in the process of proving Theorem NV2 are given in Table 13.

Table 11: Indirect Write Statements in *Vfprintf()* Source Code

```
L1:   buf[n]=data%10+'0';
L2:   *(int*)ap = count ;
```

Table 12: Theorems Need to Prove for *Vfprintf()*

Theorem NV1A: If S1 is the store before executing Line L1, then ExpT(S1,(^ n)) = false
Theorem NV1B: If S2 is the store after executing Line L1, then LocT(S2,(buf + 10)) = false
Theorem NV2: If S3 is the store before executing Line L2, then ExpT(S3, (^ ^ ap)) = false

Table 13: Sufficient Conditions to Ensure the Validity of Theorem NV2

1. ap never points to any location within the current function frame.
2. *ap never points to the location of variable ap, i.e., *ap != &ap.
3. Suppose the memory segment that ap sweeps over is called *ap_activitiy_range*, no locations within *ap_activitiy_range* are tainted before *Vfprintf()* is called.
4. *ap never points to any location within *ap_activitiy_range*.

The four conditions form a set of sufficient conditions, which if satisfied, guarantee that there is no pointer taintedness situation in the analyzed version of *Vfprintf()*. Format string vulnerabilities do not satisfy condition 3 (Table 13). As illustrated in Figure 1, the tainted data (word 0x08049978) is located in the activity range of ap, i.e., ap points to this data. For the other three conditions, we are currently unaware of any existing applications violating them.

6. EXAMPLES ILLUSTRATING VIOLATIONS OF LIBRARY FUNCTIONS' PRECONDITIONS

In the previous section, we have given a significant number of preconditions for common library functions. Not all of them are likely to occur in real application code. In this section, we give possible scenarios (constructed examples) in which some of the preconditions detailed in the previous section are violated, and explain how an attacker can exploit them. To the best of our knowledge, these vulnerabilities have not been reported in any real application or described in the literature.

6.1 Example of *strcpy()* violation – condition 3

Condition 3 in Table 7 for *strcpy()* states that the buffer *dst* does not cover the function frame of *strcpy()*, which consists of *dst, src* and *res.* Otherwise it is possible to overwrite the stack frame of *strcpy()* and modify the address of the *dst* string. Since *strcpy()* can write to the location *(*dst),* this can be used to write to any memory location, including function pointers, and hence transfer control to malicious code.

Consider the code sample in Table 14a, in which *buf* and *input* are allocated on the stack in the function frame of *foo()*. The string *input* is obtained from the user and passed as the *src* argument of *strcpy().* The *dst* argument of *strcpy()* is *buf* + *index*, where *index* is computed by subtracting the length of *input* from the end of the buffer *buf*. After *strcpy()* is called, the stack frame looks as shown in Table 14b. Assume that the attacker enters an input string longer than 20 bytes as input. Since the input buffer has a size of 100 bytes, this may not cause buffer overflow. However, this makes the value of *index* computed to become negative, which in turn makes *dst* point to a stack location before *buf* and in the function frame of *strcpy()* (thereby violating the pre-condition). In the above example, setting the *index* to (-16) makes *dst* point to the location of itself on the stack. The *strcpy()* code then writes to the location of *(*dst)*, thereby overwriting *dst* itself. Subsequent writes to *(*dst)* can then modify the contents of the location pointed to by this new value of *dst*. This allows the attacker to write any value to any memory location, including sensitive locations such as function pointers.

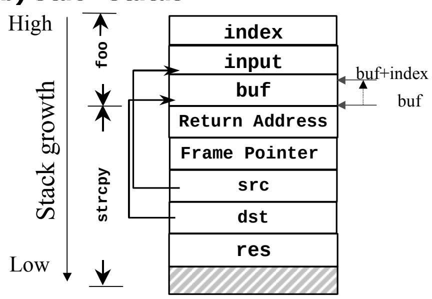
The functionality of the code shown in Table 14a is to push data to the end of a buffer. We believe it is possible that applications require such a functionality. For example, a program may need to copy data at the end of a buffer and prefix headers in front the data. The pointer arithmetic shown in Table 14a is an efficient means of implementing such an operation, so we argue that the sample code demonstrates a possible scenario in real applications.

Table 14: Violation of condition 3 of *strcpy()*

a) Sample Code

```
void foo( ) {
  int index;
  char input[100];
  char buf[20];
  scanf("%s", input );
  index = 20 –  strlen( input );
  strcpy( buf + index ,  input );
}
```

b) Stack Status



6.2 Example of *strcpy()* violation – condition 2

In Table 7, condition 2 of *strcpy()* states that *src* and *dst* do not overlap in such a way that *dst* covers the null-terminator of *src,* otherwise the null-terminator of *src* string gets overwritten and the program can go into an infinite loop. This can happen in two ways: by a buffer overflow error or by an inadvertent free error, as illustrated in Table 15a and Table 15b, respectively.

Table 15: Examples of *strcpy()* condition 2 violations

a) Buffer Overflow Error	**b) Inadverent Free Error**
char* src = malloc(20);	src = malloc(40);
char* dst = malloc(20);	snprintf(src, 30, "some string of 30 or more characters");
sprintf(src,"string with > 20 characters");	free(src);
strcpy(dst, src);	foo = malloc (10);
	dst = malloc(20);
	strcpy(dst, src);

In the first piece of code (Table 15a), two buffers are allocated on the heap and one of them is overflowed. This buffer is then passed as the *src* argument to *strcpy()*, and the other one as the *dst* argument. Upon running this code multiple times[3], we found that the memory manager consistently allocated nearly consecutive, successive memory addresses to *src* and *dst* respectively. As a result, when the *src* buffer is overflowed, its contents spill

[3] We tried it with glibc on x86-linux and Sun Solaris platforms. Our results indicate that this is not an OS or platform specific phenomenon, but a feature of glibc.

into the *dst* buffer, causing it to overlap with the *src* string and cover the null-terminator, leading to a violation of the pre-condition.

The *src* and *dst* arguments can also overlap if the destination buffer is allocated from some portion of the source buffer. This situation is illustrated in Table 15b. Here *src* is first allocated on the heap and then freed, which returns the *src* buffer to the free pool. When *malloc()* requests are made subsequently for *foo* and *dst*, the memory manager reuses the block most recently returned to it, namely the *src* buffer, for allocating the buffers *foo* and *dst*. When *strcpy()* is called, *src* and *dst* overlap in such a way that *dst* covers the null terminator of *src*, which is a violation of the pre-condition. In real codes, this can happen as a result of using a buffer that is freed on an infrequently executed path and may not be uncovered during testing.

6.3 Example of *free()* violation – condition 2

Condition 2 of the *free()* function in Table 10 states that the pointer passed to *free()* must be within the heap range. This arises from the fact that the *free()* function itself does not perform this check. When a block is freed, the *free()* function checks for an integer value at the beginning of the block, which represents the size of the block to be freed. If it finds such an integer, it does the free, irrespective of whether the block is on the heap or not.

Consider what happens when a local buffer on the stack is passed to the *free()* function in Table 16. In this code, the local array *buf* of function *foo()* is passed to the function *print_str(),* which checks if the length of the string passed to it is more than the value *n* specified by the user, and if so, frees the buffer. The pointer p which is freed by *print_str()* is aliased to *buf,* which is allocated on the stack in the function frame of *foo()*, leading to a violation of the pre-condition. Since this happens only when the user enters a string of more than 50 characters, it may not be uncovered while testing. In this example, the integer *i*, which is a local variable of *foo()* is present on the stack at the beginning of the block *buf*. The *free()* function assumes that this is the size of the buffer *buf* and attempts to deallocate a block of that size. Since the user also supplies this value, it is possible to free a block of any arbitrary size on the stack, and overwrite the contents of any memory location.

Table 16: Violation of condition 2 of *free()*

void foo() {
    char buf[100];
    int i;
    scanf("%d", &i);
    scanf("%s", buf);
    print_str(buf , 50); }
	void print_str(char* p, int n) {
	    if (strlen(p) > n) {
	        free(p);
	        return;
	    }
	    printf(stdout, "%s", p); }

7. CONCLUSIONS AND FUTURE DIRECTIONS

This paper is motivated by a low level analysis of various security vulnerabilities, which indicates that the common cause of the vulnerabilities is pointer taintedness. To reason about pointer taintedness, a memory model is needed. The main contribution of this paper is the equational definition of a memory model, which associates the *taintedness* attribute with memory locations rather than with program symbols. Pointer taintedness analysis is applied on several C library functions to extract security preconditions. The results show that pointer taintedness analysis can expose different classes of security vulnerabilities, leading us to believe that pointer taintedness provides a unifying perspective for reasoning about security vulnerabilities. We plan to extend this work in a number of ways: (1) Reduce the amount of human intervention in the theorem proving tasks; (2) Define semantics for other C statements, such as `goto`, `break` and `continue` to make the technique more widely applicable; (3) Incorporate this technique into compiler-based static checking tools to extend to large applications.

Acknowledgments:

We thank Jose Meseguer for his insightful suggestions on the project, and Tammi O'Neill for her careful reading of an early draft of this manuscript.

References:

[1] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. In IEEE Software, Jan/Feb 2002

[2] B. Chess. Improving Computer Security Using Extended Static Checking. IEEE Symposium on Security and Privacy 2002

[3] J. A. Goguen and G. Malcolm. Algebraic Semantics of Imperative Programs. MIT Press, 1996, ISBN 0-262-07172-X

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott *The Maude 2.0 System.* In Proc. Rewriting Techniques and Applications, 2003, 2003.

[5] J. Xu, Z. Kalbarczyk and R. K. Iyer. Transparent Runtime Randomization for Security. To appear in Proc. Symposium on Reliable and Distributed Systems, 2003.

[6] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. Network and Distributed System Security Symposium (NDSS2000).

[7] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format String Vulnerabilities With Type Qualifiers. 10th USENIX Security Symposium, 2001.

[8] C. Cowan, C. Pu, D. Maier, et al. Automatic Detection and Prevention of Buffer-Overflow Attacks. 7th USENIX Security Symposium, San Antonio, TX, January 1998.

[9] A. Baratloo, T. Tsai, N. Singh, Transparent Run-Time Defense Against Stack Smashing Attacks, Proc. USENIX Annual Technical Conference, June 2000.

[10] S. Chen, Z. Kalbarczyk, J. Xu, R. K. Iyer. "A Data-Driven Finite State Machine Model for Analyzing Security Vulnerabilities". in IEEE International Conf. on Dependable Systems and Networks, 2003.

[11] Introduction to equational logic. http://www.cs.cornell.edu/Info/People/gries/Logic/Equational.html

[12] S. Chen, K. Pattabiraman, Z. Kalbarczyk, R. K. Iyer. Formal Reasoning of Various Categories of Widely Exploited Security Vulnerabilities By Pointer Taintedness Semantics (Full Version). http://www.crhc.uiuc.edu/~shuochen/pointer_taintedness