Computer Vulnerability Analysis

Ivan Krsul, Eugene Spafford, Mahesh Tripunitara
Technical Report COAST Laboratory*
Purdue University
West Lafayette, IN 47907–1398
{krsul,spaf,tripunit}@cs.purdue.edu

May 16, 1998

Abstract

Many engineering fields have recognized the need to analyze the past in hope of learning from past mistakes and failures. In computer science this realization has resulted in the development of software testing techniques that attempt to detect known problems from software systems and in improved compilers and development tools. However, there exists a series of software failures where detailed analysis is rarely published, mainly for fear that the information could be used against active systems. These software failures, commonly referred to as *computer vulnerabilities*, have special properties that set them apart from traditional software failures. Detailed analysis of the factors that contribute to the existence of these vulnerabilities is mostly limited to cryptic articles posted to hacker newsgroups or web sites. There are a few notable exceptions, and this report attempts to add to these with a detailed analysis of five common computer vulnerabilities. The analysis of each vulnerability identifies its characteristics, the [expected] policies violated by its exploitation, and contributes to the understanding of the steps that are needed for the eradication of these vulnerabilities in future programs.

1 Introduction

Many engineering fields have recognized the need to analyze the past in hope of learning from past mistakes and failures [FC97, LS92, Pet85, Per84, Dor96]. In computer science this realization has resulted in the publication of case-studies in software failures [MFS90, MKL^{+}95, End75, BP84, OW84, Mye76, Knu89], in the development of software testing techniques that attempt to detect known problems from software systems [Spa90, How76, How78, JK80, Mye79, DMMP87, Lip79, Y^{+}85, WC80, Tai89, ADHe89, Bei83, DM91], and in improved compilers and development tools.

However, there exists a series of software failures where detailed analysis is rarely published, mainly for fear that the information could be used against active systems. These software failures, commonly referred to as *computer vulnerabilities*, have special properties that set them apart from traditional software failures. They allow an external subject to trigger the failure and they normally result in violation of [expected] policies. Detailed analysis of the factors that contribute to the existence of these vulnerabilities is mostly limited to cryptic articles posted to hacker newsgroups or web sites.

There are a few notable exceptions [Lin75, Spa89a, Spa89b, Sto90, Kum95, DFW96, MF97, DW95], and this report attempts to add to these with a detailed analysis of four common computer vulnerabilities. The analysis of each vulnerability identifies its characteristics, the [expected] policies violated by its exploitation, and contributes to the understanding of the steps that are needed for the eradication of these vulnerabilities in future programs.

Where appropriate, we use the notation introduced in [KST98] for the formal specification of the expected policies violated by the exploitation of vulnerabilities. This notation is part of a model that can be used to represent high-level policies as mathematical expressions. The model takes a functional approach to the specification of policies that allows the stepwise refinement of policies, such that at higher levels we deal with abstractions and at lower levels with

*Portions of this work were supported by sponsors of the COAST Laboratory.

details. The model takes advantage of the natural hierarchical organization of computer systems, with systems being composed of objects with attributes.

This policy specification model assumes that policies are a set of rules that define the acceptable *value* of a system, as defined by its owners, as its state changes through time. Policies are represented as an aggregation of the value of the components, where the value of components is an aggregation of the value of its sub-components. In this model we define a function *Policy* that takes as parameters the state of a system before and after an atomic operation. The function will return a value of `true` or `false` depending on whether the operation has violated policy because the value of the system changed from one state to the next. The *Policy* function needs a helper function, called the *System Value* function, that calculates the value of the system at a particular state, and it calculates the value of that system by aggregating the value of its objects. Further details and examples on this notation can be found in [KST98].

2 Buffer Overflows

The so-called *buffer overflow* vulnerability is difficult to characterize. There are many variations, and figure 1 shows three possible variations. A program tries to copy some data from one object into another, does not check that the destination object is large enough to contain the source object, and uses a memory-copy routine such as `sprintf` to perform the copying.

[a]

Form 1

```
main(int ac,
  char *av[]) {
  p(av[1]);
}

void p(char *a){
  char b[30];

  strcpy(b,a);
}
```

Form 2

```
main() {
  p();
}

void p(){
  char b[30];
  char *p;

  p = getenv("TERM"));
  sprintf(b,"%s",p);
}
```

Form 3

```
main() {
  p();
}

void p(){
  struct hostent *h;
  sockaddr_in s;

  h = gethostbyname(*host);
  bzero(&s, sizeof s);
  s.sin_family =
       h->h_addrtype;
  s.sin_port = 25;
  bcopy(h->h_addr_list[0],
        &s.sin_addr,
        h->h_length); [1]
}
```

[a] Normally `h->h_length` would be the same size as `h->h_addr_list[0]`. However, it is possible to create a (possibly fake) DNS reply that will violate this assumption.

Figure 1: The so-called “buffer overflow” vulnerabilities.

Not all program that share this characteristic are vulnerable. The programs shown in figure 2 all have buffer overflows but are not vulnerable because either the function never returns—in which case the program never has the opportunity to jump to the code inserted—or the programs buffer is declared static—in which case the program overwrites heap and not the stack.

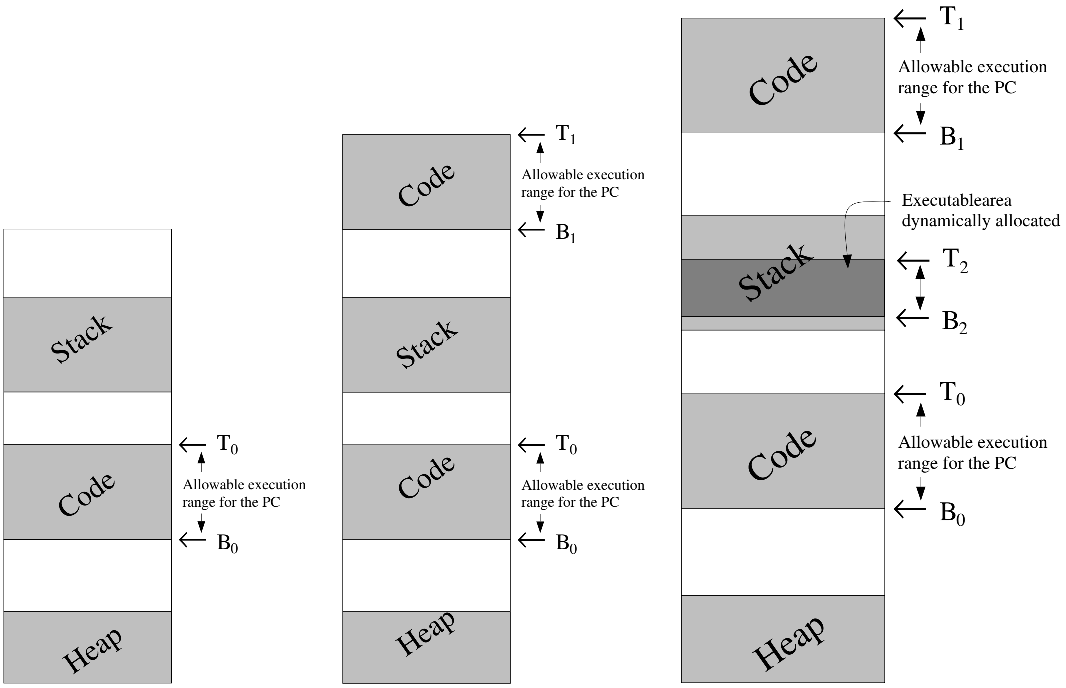
Programmers make assumptions about the environment where the code will be executed. For most architectures, programmers make the implicit assumption that the programming counter (PC) will execute the code intended by the programmer, and nothing else. Figure 3 illustrates the ranges where the PC executes in the cases of non-fragmented code segments, fragmented code segments, and fragmented code segments with dynamic loading of code[2].

The so-called “buffer overflow” vulnerabilities are instances of programs where the programming counter jumps from an allowable executable region to a region in memory, normally the stack, where it executes some arbitrary code. Because there are instances of buffer overflows that cannot be characterized as vulnerabilities, the real issue behind these vulnerabilities is not the buffer overflow but rather what happens when a user can cause the stack pointer to change so that it points back at the stack.

[2] This simplified model does not take into account the system area of memory and this can be incorporated by adding an additional set of segment market to signal that it is OK for the PC to execute system memory.

Line	*Form 1*	*Form 2*
1	`main(int ac,`	`main() {`
2	`  char *av[]) {`	`  p();`
3	`  p(av[1]);`	`}`
4	`}`	
5		`void p(){`
6	`void p(char *a){`	`  static char b[30];`
7	`  char b[30];`	`  char *p;`
8		
9	`  strcpy(b,a);`	`  p = getenv("TERM"));`
10	`  exit(1);`	`  sprintf(b,"%s",p);`
11	`}`	`}`

Figure 2: Buffer overflows that are not considered vulnerabilities.



A program normally contains three segments that can be distinguished: heap, code, and stack. We expect the PC to remain in the area between the B_0 and T_0 markers.

If the code segment is fragmented, the program counter is expected to remain in the areas between the B_0, T_0, B_1, and T_1 markers.

Dynamic code can be loaded into the stack and the program explicitly (by calling a function that dynamically links the code) creates another area where the PC is allowed to execute: the are between the B_2 and T_2 markers.

Figure 3: Programs normally execute code from well defined regions in memory, even if the memory is fragmented or the program contains dynamic executable code.

The program shown next illustrates how a program that can provide an attacker an index into arbitrary stack memory can be vulnerable to the same problem without overwriting the program's local memory or altering anything other than the return address in the stack and the portion of memory that will be used to store the code to be executed.

The program segment was extracted from a project for graduate operating system course and its function is to allow the programmer to change the value of debugging flags without having to recompile the code.

```
main() {
  int  dbg1, dbg2, dbg3, dbg4;
  int  numiter,index, i, j;
  FILE *fp;

  /* Read from a file the values of the debugging variables
     as a series of (position, value) pairs: (1,5) would set
     dbg1 to 5 and (4,0) would clear the dbg4 flag. */
  if((fp = fopen("conf","r"))!=NULL){
    /* How many flags to change? */
    fscanf(fp,"%d",&numiter);
    printf("numiter = %d\n",numiter);
    for(i=0;i<numiter;i++) {
      fscanf(fp,"%d%d",&index,&j);
      *(&dbg1-index+1) = j;
    }
  }
}
```

The expected policy violated by this example, and all of the "buffer-overflow" vulnerabilities we have seen to date, is that the PC should remain within a well defined *allowable* range. A policy specification for this case can now be generated with the model and notation presented in [KST98].

For simplicity, an atomic operation will be axiomatically defined as the execution of any instruction that causes the program counter (PC) to change. Note that, however, an alternative definition could consider only those operations that cause the program counter to jump or we could consider only those operations that cause a program to return from a subroutine. Although less general, this definition is more practical.

Our policy function, shown in equation 1, takes as arguments a system value function, an object value function, and two sets of interest (before and after the execution of an instruction). The function returns `true` if the policy has not been violated and `false` otherwise.

The policy we will specify requires that applications only execute instructions within the bounds defined. The set of interest consists of programs, program counters, and boundaries:

Programs:
- ◇ Set of boundaries: b.
- ◇ Set of program counter locations: pc.

Boundary:
- ◇ Top of allowed segment: T.
- ◇ Bottom of allowed segment: B.

Program Counter:
- ◇ Location: l.

The value functions that can be used to implement the desired policy are shown in equations 2 and 3.

2.1 Static Buffer Overflows

There is a class of vulnerabilities that results from buffer overflows that cannot be caught by the violation of the policy specified in the preceding section. We will show next an example of such a vulnerabilities—-at this point theoretical because we have no evidence that it actually exist in released systems.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{Policy} : \mathit{System\ Value\ function} \times \mathit{Object\ Value\ function} \times} \\
 & \quad \textbf{set of interest} \times \textbf{set of interest} \rightarrow \textbf{boolean} \\
\textbf{fun} & \mathit{Policy}\left(\mathit{Value}, v, I_i, I_{i+1}\right) ::= \\
 & \textbf{if } \mathit{Value}\left(I_i, v\right) \leq \mathit{Value}\left(I_{i+1}, v\right) \quad \textbf{then} \\
 & \quad \mathit{Policy} := \text{true}; \\
 & \textbf{else} \\
 & \quad \mathit{Policy} := \text{false}; \\
 & \textbf{fi} \\
\textbf{nuf} &
\end{array}
\tag{1}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{Value} : \textbf{set of interest} \rightarrow \textbf{integer}} \\
\textbf{fun} & \mathit{Value}\left(S\right) ::= \\
 & \quad \mathit{Value} := \sum_{x \in S} v\left(x, S - x\right) \quad \forall x \in S; \\
\textbf{nuf} &
\end{array}
\tag{2}
$$

The program shown next declares all its variables to be `static` and hence cannot have a buffer overflow that overwrites the stack:

```
main() {
  /* Static variables. Can't inject anything into the stack */
  static char name[10], term[5], userID[10];

  /* Do something that will determine the name and userID of the user */
  strcpy(userID,"gollum");
  strcpy(name,"peter");

  /* The program needs to know the terminal type... so lets read
     it from the environment variable */
  /* OVERFLOW HAPPENS HERE! */
  strcpy(term,getenv("TERM"));

  /* Now that we know the terminal... */
  if(strcmp(userID,"root")==0) {
     /* Do something super restricted */
     do_secret(term);
  } else {
     /* Print error message telling the user he does not have access */
     print_error(term);
  }
}
```

If the environment variable `TERM` is set to the value "vt100root" then the program will execute the function `do_secret` regardless of the original value of the variable `userID`. The program counter remains in the area allowed by design but violates the semantic of the program specification.

We argue that, although this vulnerability is the result of a buffer overflow, it belongs to a different class of vulnerabilities because the attacker cannot inject arbitrary code that will be executed by the program. Rather, the attacker can just cause existing code to be executed in a different order that specified in the program design.

3 Solaris `_iob[]` Buffer Overflow

A common fix or method for preventing traditional buffer overflows is declaring `static` buffers under the assumption that a buffer overflow would overwrite the static heap and hence would not allow exploiters to insert arbitrary code into the stack after changing the return address of the function.

In some programs (verified in Solaris 2.5 only but thought to apply to other platforms), however, it is still possible to inject and execute arbitrary code by overwriting a static buffer.

The exploitation of the vulnerability takes advantage of the fact that the `_iob[]` array—that contains information about the standard input, output, and error file descriptors—can be replaced by overwriting static buffers. Any program that uses either `stdout` or `stderr` to display information *after a static buffer overflow* can overwrite arbitrary memory with this information.

v : **object of interest** $\times$ **set of object of interest** $\rightarrow$ **integer**

fun $v\,(o, S)$ ::=
    $v := 0;$
    if o is a program **then**
        $\forall x \in o.pc$ **do**
            $m := 0;$
            $\Rightarrow$ *Check to see if the PC is in a correct range* $\Leftarrow$
            $\forall y \in o.b$ **do** (3)
                $m := 1$ **if** $x.l \geq y.B \wedge x.l \leq y.T;$
            od
            $\Rightarrow$ *Violation if we did not find a valid range for the PC* $\Leftarrow$
            $v := v - 1$ **if** $m = 0;$
        od
    fi
nuf

Consider the program shown next. It contains a static buffer that can be overflowed and, because the buffer is not in the stack, only static data can be replaced. The symbol table for this program can be generated as shown in figure 4.

```
#include <stdio.h>

main(int argc, char *argv[]) {
  static char buffer[100];

  strcpy(buffer, argv[1]);
  fprintf(stderr,"Error: %s\n",buffer);
  exit(0);
}
```

```
$ nm test | sort -n +2 +1
[Some infomration deleted]
[50]    |    133100|       0|OBJT |GLOB |0    |11     |_GLOBAL_OFFSET_TABLE_
[54]    |    133112|       0|OBJT |GLOB |0    |12     |_DYNAMIC
[59]    |    133248|       0|OBJT |GLOB |0    |13
|_PROCEDURE_LINKAGE_TABLE_
[51]    |    133296|       0|FUNC |GLOB |0    |UNDEF  |atexit
[52]    |    133308|       0|FUNC |GLOB |0    |UNDEF  |exit
[56]    |    133320|       0|FUNC |GLOB |0    |UNDEF  |_exit
[60]    |    133332|       0|FUNC |GLOB |0    |UNDEF  |strcpy
[65]    |    133344|       0|FUNC |GLOB |0    |UNDEF  |fprintf
[30]    |    133360|       0|OBJT |LOCL |0    |15     |__CTOR_LIST__
[34]    |    133360|       0|OBJT |LOCL |0    |14     |force_to_data
[41]    |    133360|       0|OBJT |LOCL |0    |14     |force_to_data
[44]    |    133364|       0|OBJT |LOCL |0    |15     |__CTOR_END__
[31]    |    133368|       0|OBJT |LOCL |0    |16     |__DTOR_LIST__
[43]    |    133372|       0|OBJT |LOCL |0    |16     |__DTOR_END__
[37]    |    133376|     100|OBJT |LOCL |0    |17     |buffer.2
[58]    |    133376|       0|OBJT |GLOB |0    |16     |_edata
[47]    |    133480|       4|OBJT |GLOB |0    |17     |_environ
[57]    |    133480|       4|OBJT |WEAK |0    |17     |environ
[49]    |    133488|     320|OBJT |WEAK |0    |17     |_iob
[55]    |    133488|     320|OBJT |GLOB |0    |17     |__iob
[48]    |    133808|       0|OBJT |GLOB |0    |17     |_end
```

Figure 4: Symbol table for the vulnerable program.

The buffer starts in memory location 133376 and the `_iob[]` array starts in memory location 133488. It is clearly possible to overwrite the `_iob[]` array by overflowing the static character array `buffer`. It is just a matter of overwriting these buffers with useful information. Each element of the `_iob[]` buffer is a structure that has the following form:

```
typedef struct {
 int            _cnt;   /* number of available characters in buffer */
 unsigned char   *_ptr;  /* next character from/to here in buffer */
 unsigned char   *_base; /* the buffer */
 unsigned char   _flag;  /* the state of the stream */
 unsigned char   _file;  /* UNIX System file descriptor */
} FILE;
```

To overwrite the `_iob[]` array we will have to overwrite the environment pointer in memory location 133480. As shown below, we can obtain the value of this pointer, which is the same for every run of the program, by using a debugger. The exploit script can be tailored to insert this value into the appropriate location:

```
$ gdb test
(gdb) break main
Breakpoint 1 at 0x106f8: file test.c, line 6.
(gdb) run
Starting program: /tmp/test

Breakpoint 1, main (argc=1, argv=0xeffffaac) at test.c:6
6         strcpy(buffer, argv[1]);
(gdb) printf "0x%x\n",_environ[0]
0xeffffbc2
```

It should be clear now that the `_iob[]` array can be replaced with arbitrary values. The significance of this is that all print routines that use standard output and standard error write to the memory location indicated by the corresponding `_iob[]` array element. Hence, by overwriting these values—and providing that we can predict that the program vulnerable will print an error message with text we can inject—we can overwrite arbitrary portions of memory with arbitrary information. This can be used to insert executable code in static memory that can be executed by the program.

Note that the vulnerable program calls the `exit` system call after it prints a message that echoes the first argument given. In many versions of UNIX, including Solaris 2.5, this routine is dynamically linked and hence its address must be resolved at runtime. The symbol table for the vulnerable program (see figure 4) shows that the address of `exit` is undefined at compile time. In fact, the location of the exit system call is normally referenced as an offset of the symbol _PROCEDURE_LINKAGE_TABLE_. As shown next, this offset can be obtained by using a debugger.

```
$ gdb test
(gdb) break exit
Breakpoint 1 at 0xef7747bc
(gdb) run
Starting program: /tmp/test
Breakpoint 1, 0xef7747bc in exit ()
(gdb) disassemble exit
Dump of assembler code for function exit:
0xef7747bc <exit>:      call  0xef7915fc <_PROCEDURE_LINKAGE_TABLE_+7176>
0xef7747c0 <exit+4>:    nop
0xef7747c4 <exit+8>:    mov  1, %g1
0xef7747c8 <exit+12>:   ta  8
End of assembler dump.
```

Hence an exploit script can overwrite this location in memory, that does not change from run to run, and when the original program calls the `exit` system call the code injected would be executed instead.

Naive exploit scripts released in the Internet simply insert the assembly code to be executed in the space between the symbol of the `exit` call and the beginning of the data section. Generally it is possible to insert sufficient assembly code here to start a shell. However, since arbitrary locations of memory can be changed, it is possible to change the code of the program itself.

4 IP Fragmentation

In this section, we describe and analyze a vulnerability in the Internetworking Protocol (IP). An interesting characterstic of this vulnerability is that it is inherent to the design of the protocol and not a particular implementation. In section 5, we examine a vulnerability that also involves IP fragmentation, called *teardrop*. Although both vulnerabilities involve IP fragmentation, our analysis reveals that the violated policies are different for the two of them, and therefore, the two vulnerabilities are different from one another.

4.1 IP and IP Fragmentation

IP is the central protocol of the TCP/IP suite of protocols[Pos81a]. IP provides for unreliable, connectionless datagram oriented communication services. An IP datagram is constructed from a *Protocol Data Unit* (PDU) or *packet* from a higher layer protocol such as the Transmission Control Protocol (TCP), by prepending a *header* to the packet [Pos81b]. The packet from the protocol at a higher layer is then the *data* of the IP datagram. The header contains control information, such as the length of the datagram. The reader is referred to [Com95] for more details on IP and its role in the TCP/IP protocol suite.

An IP datagram has a maximum size of 64 kBytes. This is called the *Maximum Transmission Unit* (MTU) of IP. Thus, a protocol at a higher layer in the TCP/IP protocol stack, that uses IP's communication services, must ensure that a packet of size at most 64 kBytes minus the length of the IP header is handed to the IP layer for transmission. Similarly, the protocol at the layer below IP whose communication services are used by IP, also has an MTU, which may be smaller than IP's MTU. Therefore, before an IP datagram can be transferred to the protocol at the lower layer, it may need to be *fragmented*.

Some or all of these fragments arrive at the destination, perhaps out of order, where they are *reassembled*. Fragments of an IP datagram are very similar in structure to the original datagram, that is, they consist of an IP header and data. In fact, an IP datagram that is not fragmented is equivalent to the first (and only) fragment of itself. We refer the reader to the [Pos81a] for more details on fragmentation and reassembly.

The following four fields of the IP header provide sufficient information to reassemble datagrams [Pos81a].

Identification: This field is the same for all the fragments of a single datagram. The identification field, in conjunction with the source address, destination address and protocol number, is used to identify all the fragments of a single datagram in the internet at any given time.

Length: The length field contains the length of each fragment.

Flags: One of the bits in the 3-bit flag is a *more–fragments* bit. If it is set for a particular fragment, then that fragment is not the last fragment of a datagram. For the last fragment of a datagram, the more–fragments bit is zero.

Fragment Offset: The fragment offset for a particular fragment indicates the position of the first byte of the data it carries, in the data of the entire datagram. The fragment offset is in units of 8 bytes. Thus, the data portion of any IP fragment must be at least 8 bytes in size.

4.2 Description of the Vulnerability

The vulnerability we describe here is one considered in [ZRT95][3]. The vulnerability is in the reassembly process as described in [Pos81a].

It is possible that fragments overlap each other when they arrive at the destination. [Pos81a] states that a fragment should overwrite portions of fragments that arrived earlier, that overlap it. If the data in a datagram is that of a protocol that includes "sensitive" information for that protocol, it is possible for portions of such information to be carried in two different, overlapping fragments. And the information may be different in each of the fragments. For instance, TCP packets have a header that includes a field called *syn*, which, if set, indicates a request for a connection. It is possible for an IP fragment to carry the portion of the TCP packet that corresponds to the *syn* bit as zero, and another to carry this as one. If the latter arrives later, the packet passed to TCP by IP would correspond to a connection request. If the former arrives later, it would not.

A vulnerability is exists because there is a need to make inferences about the data carried by an entire datagram before it is reassembled at the destination's IP layer. An instance of where this is needed is a packet filter, that attempts to enforce access rules on communication traffic [ZRT95]. A packet filter, situated on a path between the source and destination, might want to disallow TCP connection requests from going through. Packet filters that do not maintain state across fragments of a datagram decide on the fate of a datagram (let through or drop) by imposing the access rules on the fragment that carries an IP header with a fragment offset of zero, which is the first fragment of a datagram. Fragments with a non-zero fragment offset are allowed to proceed to their respective destinations. Without a fragment with an offset of zero, reassembly cannot be completed.

[3] [ZRT95] also discusses the problem of "tiny fragments", which we do not consider in this paper

Thus, a packet filter configured to drop TCP connection requests would allow a fragment, that carries a TCP packet with its *syn* bit set to zero, and has an offset of zero, go through. It would also allow a fragment, that carries a TCP packet with its *syn* bit set, but that has a non-zero offset, to go through. If the first fragment arrives earlier than the second, the second would overwrite the first at the destination. Thus, the datagram, when fully assembled would correspond to a TCP connection request.

While our interest is in analyzing this vulnerability and not in suggesting solutions, we discuss some of the proposed solutions to gain more insight into the vulnerability. [Mog89] suggests a packet filter that maintains state across fragments to solve the above problem. But, such a packet filter is unable to handle the case where the fragments with a non-zero offset arrive before that with an offset of zero.

[ZRT95] proposes a packet filter that drops fragments that do not have an offset of zero, but have an offset that coincides with "sensitive" portions of the data. For instance, a TCP header is 20 bytes long, of which the last 4 bytes are the checksum and an urgent pointer, neither of which is deemed useful to a packet filter. The first 16 bytes of the TCP header can only be split across fragments by having the fragment with an offset of zero only contain the first 8 bytes, the minimum size for a fragment. The fragment with an offset of 1 would carry the next 8 bytes. The packet filter drops any fragment with an offset of 1, carrying TCP traffic.

[WS95] proposes a solution in which the reassembly routine gives data from lower offset fragments precedence in the case of overlaps, immaterial of the order of arrival of the fragments. This solves the problem in the specific case of applying access rules to TCP connection requests, for instance, but is not in keeping with the IP standard [Pos81a]. Also, it assumes that data carried by a lower offset fragment is the "correct" one.

4.3 An Analysis of the Vulnerability

Program 1

```
main() {
  int i, j;

  i = j = 0;

  *(&i - 1) = 1;

  if(j != 0) {
    /* Do sensitive stuff */
  }
}
```

Program 2

```
main() {
  int i, j;

  i = j = 0;

  *(&i - 1) = 0;

  if(j != 0) {
    /* Do sensitive stuff */
  }
}
```

Program 1 results in a security violation, while Program 2 does not.

Figure 5: Examples related to the IP Fragmentation problem.

We compare the problem of IP fragmentation described above to the program listed as program 1 in figure 5. In program 1, the variable `j` is accidentally (or purposefully) overwritten such that it causes a security violation. That the environment (which includes, for instance, the compiler) allows for such an indirect change in `j` represents a vulnerability. In program 2, on the other hand, despite the existence of the vulnerability, no security violation occurs.

At the time the violation occurs, it is not always possible to trace back to the variable that caused it. Also, at the time the violation occurs, it is not possible to determine whether the content of the variable has been changed in an unauthorized manner. But, in the case of IP fragmentation, the "sensitive" storage locations are known *a priori*, before the processes of fragmentation and reassembly take place.

The violated policy is: *Data in overlapping portions of fragments must be identical to each other*. Thus, in the case of program 1 from figure 5, we require that any unauthorized changes to `j` be such that the value of `j` before and after the change be the same, that is, that the change have no effect.

An assumption made when IP fragments are constructed and sent is that data in the overlapping portions of fragments are identical to each other, since the fragments belong to the same datagram. During reassembly, an exploitation

$$
\begin{array}{l}
v : \textbf{object of interest} \times \textbf{set of objects of interest} \rightarrow \textbf{integer} \\
\textbf{fun} \quad v\,(o, A) ::= \\
\qquad v := 0; \\
\qquad \forall \bar{o} \in A \ \textbf{do} \\
\qquad\qquad \textbf{if } o.s \leq \bar{o}.s < o.s + o.n \textbf{ then} \\
\qquad\qquad\qquad i := \bar{o}.s - o.s; \\
\qquad\qquad\qquad m := min\{\bar{o}.n, o.n - i\}; \\
\qquad\qquad\qquad v := v - 1 \textbf{ if } (o.b_{i+1} \neq \bar{o}.b_1) \vee \ldots \vee (o.b_{i+m} \neq \bar{o}.b_m)\,; \\
\qquad\qquad \textbf{elseif } \bar{o}.s \leq o.s < \bar{o}.s + \bar{o}.n \textbf{ then} \\
\qquad\qquad\qquad i := o.s - \bar{o}.s; \\
\qquad\qquad\qquad m := min\{o.n, \bar{o}.n - i\}; \\
\qquad\qquad\qquad v := v - 1 \textbf{ if } (\bar{o}.b_{i+1} \neq o.b_1) \vee \ldots \vee (\bar{o}.b_{i+m} \neq o.b_m)\,; \\
\qquad\qquad \textbf{fi} \\
\qquad \textbf{od} \\
\textbf{nuf}
\end{array}
\tag{4}
$$

of the vulnerability is indicated by overlapping portions of fragments being different from one another. Thus, our set of interest is all the data that needs to be written into storage locations.

Data:
◇ The number of bytes that need to be stored, n.
◇ The sequence of n bytes that need to be stored, $b_1, b_2, \ldots, b_n$.
◇ A location s, which is the starting location where the data is to be stored.

The policy function we adopt is the same as that in equation 1. We require that the function $Policy$ be evaluated after the execution of a list of instructions that constitutes an atomic operation. A list of instructions that contains a write instruction of any of the data items of interest as the last instruction, forms an atomic operation.

The object value function v and the system value function $Value$ are given by equations 4 and 2 respectively. The object value function lowers the value of an object (a piece of data) if a write operation is performed such that there is an overlap between the locations occupied by two pieces of data, and the overlapping bytes are not identical. This is turn lowers the system value function because the system value function simply performs an aggregation of the values of all the objects. This results in the function $Policy$ returning $false$ indicating a policy violation.

The policy specification we have prescribed in the functions $Policy$, v and $Value$ can be incorporated into the algorithm for reassembly of IP fragments to detect violations of the policy and reject such datagrams. In particular, they can be used to augment the algorithm given in [Cla82] for reassembly.

5 The *teardrop* Vulnerability in Linux

Teardrop is a vulnerability in the IP fragment reassembly program in Linux [Arc97]. We describe the problem in this section, and analyze it in section 5.2.

As we mentioned in a section 4.1 of this paper, IP fragmentation is used to meet the smaller MTU requirement of the protocol running below IP. When fragments arrive at a destination, they need to be reassembled. The possibility of fragments overlapping is a vulnerability in the design of IP, as discussed in section 4.

A vulnerability that appears to be related to the vulnerability discussed in section 4 was discovered in the Linux operating system [Arc97]. The system would crash when reassembling certain overlapping fragments. Though superficially this problem was characterized as an IP fragmentation problem in [Arc97], our analysis from the next section shows that the policy violated by the exploitation of this vulnerability is quite different from the problem of overlapping fragments that we discussed in a section 4.

5.1 A Description of the Problem

The program for reassembling IP fragments in Linux runs in a loop, copying payload from queued fragments into a buffer. The C code is reproduced in figure 6.

In section 1, qp is a linked list of fragments to be reassembled. The *if* condition is used to check whether the length claimed in the IP header for the fragment is greater than the length of the data received. This prevents copying of more data than received in the fragment. Finally, the data and header for the fragment are copied into the buffer pointed to by `ptr` and a variable that keeps track of the total length of the datagram, `count`, is updated, and processing proceeds to the next fragment[4].

A variable `end` is used to hold the address of the location (byte) immediately following the last byte in the fragment. Thus, `end` is computed as `end = offset + ntohs(iph->tot_len) - ihl;` for each fragment as reassembly takes place. To avoid problems caused by overlapping fragments, such as the one discussed in section 4, a check is in place to detect a fragment that overlaps with a fragment already in the reassembly buffer. The code for this is in section 2 in figure 6.

The condition in the *if* statement checks whether the offset of the fragment currently being considered in the reassembly procedure is within the previous fragment. If so, the program realigns the fragment so that its offset is the location immediately following the previous fragment by updating the `offset` variable. This is not in keeping with the IP specification since the specification calls for a fragment overlapping with another to overwrite it [Pos81a].

The length of the fragment is updated as indicated in section 3 of figure 6. A `memcpy()` function call is then issued to copy the data from the fragment into the reassembly buffer.

The problem with the program is that the variable `end` is not also adjusted when `offset` is adjusted in the case of overlaps. Thus, `fp->len` in line 3 in section 3 could be negative at the time the `memcpy()` call is issued with `fp->len` as the number of bytes that need to be copied. The type for the parameter to `memcpy()`, that denotes the number of bytes to be copied, is `unsigned int` and therefore a negative integer denotes a very large number of bytes that need to be copied, which results in a crash, since the locations referred to in the `memcpy()` call are meaningless to the process performing the reassembly.

Line	*Section 1*	*Section 2*	*Section 3*
1	`fp = qp->fragments;`	`if(prev != NULL &&`	`fp->offset = offset;`
2		`offset < prev->end)`	`fp->end = end;`
3	`while(fp != NULL) {`	`{`	`fp->len = end - offset;`
4	`if(count+fp->len >`	`i = prev->end - offset;`	
5	`skb->len) {`	`offset+= i;`	
6	`error_too_big;`	`ptr+= i;`	
7	`}`	`}`	
8			
9	`memcpy((ptr+fp->offset),`		
10	`fp->ptr, fp->len);`		
11	`count+= fp->len;`		
12			
13	`fp = fp->next;`		
14	`}`		
15			
16			
17			

Figure 6: Portions from the reassembly program in Linux

5.2 An Analysis of the Vulnerability

The problem with the code is that `end` is not updated when `offset` is updated. This is not a problem unique to IP fragmentation and reassembly. There are other instances of programs containing variables that have implicit relationships with each other. But the programming environment or language does not necessarily provide the tools to express such dependencies within the program. Also, a programmer does not always attempt to express or enforce such relationships in a program.

We provide another example of such a problem in figure 7. Section 1 shows the definition of a data type `llist` which is used to maintain linked lists. A variable `lst` is defined of that type and its fields are initialized.

The field `head`, the head of the linked list, and the field `len`, the number of elements in the linked list, are related. But the programmer does not encode or enforce this relationship in the program. Thus, in section 2, in the function that

[4]Note that reassembly has not yet taken place.

Section 1

```
struct llist {
  struct llistElem *head;
  int len;
};

struct llist lst;

lst.head = NULL;
lst.len = 0;
```

Section 2

```
void Delete(llist l,
            int numToDelete);
{
  while(numToDelete > 0)
  {
    /* Delete an item from
       the head and move
       the head forward */

    numToDelete--;
  }
}
```

Section 3

```
Print(llist l)
{
  for(int i = l.len; i > 0;
      i--) {

    /* Print the i-th
       element */

  }
}
```

Figure 7: Example showing relationships between variables not being enforced in the program

carries out the deletion of `numToDelete` entries in the list, if the `len` field is not adjusted as the `head` is changed, the program will crash when the `Print()` function from section 3 is invoked.

The violated policy is: *Relationships between variables should be encoded and enforced within the program.* Our set of interest is the set of all programs. Within the programs, we are interested in all variables. For each variable, we seek a list of other variables the variable is related to, and a function defining the relationship.

Program:

◇ Set of variables, s. Note that "variable" includes fields within compound data types such as structures in C and arrays.

Variable:

◇ Set of tuples, $t = (w_1, \ldots, w_n, f)$, where $w_1, \ldots, w_n$ are variables that the variable in question, v is related to, and f is a function that enforces the relationship. Thus, f takes as arguments the value of each of $v, w_1, \ldots, w_n$ and returns a boolean result indicating whether the relationship is satisfied or not.

The policy function we adopt is the same as that in equation 1. We require that the function $Policy$ be evaluated for a policy violation to be detected after the execution of a list of instructions that constitutes an atomic operation. In particular, we require that the policy function be evaluated before any read operation of a variable in the program. The system value function aggregates the values in all the objects in the system, that is, the function we adopt is the one specified in equation 2. The object value function v is given in equation 5. The object value function lowers the value of an object if a pre–specified relationship with other objects is not satisfied by it after an atomic operation. This in turn lowers the result returned by the system value function.

6 Conclusion

In this paper we presented a detailed analysis of four computer vulnerabilities that are representative of common errors that could be prevented or fixed if appropriate emphasis would be placed on enforcing the policies assumed by the developers. We have also shown that there is a large discrepancy between the assumptions programmers and designers make with respect to the environmental characteristics of the systems where their software will execute and the characteristics of the actual operational environment.

v : **object of interest** $\times$ **set of objects of interest** $\rightarrow$ **integer**

$$
\begin{array}{l}
\textbf{fun} \quad v\,(o,A) ::= \\
\quad v := 0; \\
\quad \forall o \in A \textbf{ do} \\
\quad\quad \textbf{if } o \text{ is a program } \textbf{then} \\
\quad\quad\quad \forall w \in o.s \textbf{ do} \\
\quad\quad\quad\quad \forall u \in w.t \textbf{ do} \\
\quad\quad\quad\quad\quad v := v - 1 \textbf{ if } f(w, w_1, \ldots, w_n) = \text{ false} \\
\quad\quad\quad\quad\quad\quad\quad \text{where } u = (w_1, \ldots, w_n, f) \\
\quad\quad\quad\quad \textbf{od} \\
\quad\quad\quad \textbf{od} \\
\quad\quad \textbf{fi} \\
\quad \textbf{od} \\
\textbf{nuf}
\end{array}
\tag{5}
$$

References

[ADHe89] H. Agrawal, R. DeMillo, R. Hathaway, and et al. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, 1989.

[Arc97] Rootshell Archive. Linux IP Fragmentation Bug - Teardrop. *http://www.rootshell.com/archive-evz8t6yapbpetxcq/199711/teardrop.c*, November 1997.

[Bei83] Boris Beizer. *Software Testing Techniques*. Electrical Engineering/Computer Science and Engineering Series. Van Nostrand Reinhold, 1983.

[BP84] V.R. Basili and B.T. Perricone. Software Errors and Complexity. *Communications of the ACM*, 27(1):42–52, January 1984.

[Cla82] David D. Clark. *RFC-815 IP Datagram Reassembly Algorithms*. Computer Systems and Communications Group, July 1982.

[Com95] D. E. Comer. *Internetworking with TCP/IP*. Prentice-Hall, third edition, 1995.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy 1996*, pages 190–200. Princeton University, 1996.

[DM91] Richard A. DeMillo and Aditya P. Mathur. On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software. Technical report, Software Engineering Research Center, Purdue University , SERC-TR-92-P, March 1991.

[DMMP87] Richard A. DeMillo, W. Michael McCracken, R. J. Martin, and John F. Passafiume. *Software Testing and Evaluation*. The Benjamin/Cummings Publishing Company Inc., 1987.

[Dor96] Dietrich Dorner. *The logic of failure: why things go wrong and what we can do to make them right*. Metropolitan Books, 1996.

[DW95] Drew Dean and Dan S. Wallach. Security Flaws in the HotJava Web Browser. Technical Report 501-95, Department of Computer Science, Princeton University, November 1995.

[End75] A. Endres. An Analysis of Errors and Their Causes in System Programs. *IEEE Transactions on Software Engineering*, SE-1(2):140–149, June 1975.

[FC97] Jacob Feld and Kenneth Carper. *Construction failure*. John Wiley, 2nd edition, 1997.

[How76] W. E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, SE-2(3):208–214, 1976.

[How78] W. E. Howden. An Evaluation of the Effectiveness of Symbolic Testing. *Software Practice and Principle*, 8(4):381–397, July-August 1978.

[JK80] White L. J and Cohen E. K. A Domain Strategy for Computer Program Testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, May 1980.

[Knu89] Donald E. Knuth. The Errors of TEX. *Software—Practice and Experience*, 19(7):607–685, July 1989.

[KST98] Ivan Krsul, Eugene Spafford, and Tugkan Tuglular. A New Approach to the Specification of General Computer Security Policies. Technical Report COAST Technical Report 97-13, COAST Laboratory, Department of Computer Sciences, Purdue University, January 1998.

[Kum95] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, August 1995.

[Lin75] Richard R. Linde. Operating system penetration. In *National Computer Conference*, 1975.

[Lip79] M. Lipow. Prediction of Software Failures. *Journal of Systems and Software*, 1(1):71–75, 1979.

[LS92] Matthys Levy and Mario Salvadori. *Why Buildings Fall Down*. W. W. Norton & Company, 1992.

[MF97] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes and Antidotes*. John Wiley & Sons, Inc., 1997.

[MFS90] B. Miller, L. Fredrikson, and B. So. An Embirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[MKL+95] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Akitkumar Natarajan, and Jeff Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical report, Computer Science Department, University of Wisconsin, November 1995.

[Mog89] Jeffrey C. Mogul. Simple and Flexible Datagram Access Controls for UNIX-based Gateways. In *USENIX Conference Proceedings*, pages 203–221, 1989.

[Mye76] Glenford J. Myers. *Software Reliability*. Wiley-Interscience, 1976.

[Mye79] G. Myers. *The Art of Software Testing*. Wiley, 1979.

[OW84] T.J. Ostrand and E.J. Weyuker. Collecting and Categorizing Software Error Data in an Industrial Environment. *The Journal of Systems and Software*, 4:289–300, 1984.

[Per84] Charles Perrow. *Normal Accidents: Living With High-Risk Technologies*. Basic Books, 1984.

[Pet85] Henry Petrosky. *To engineer is human : the role of failure in successful design*. St. Martin's Press, 1985.

[Pos81a] J. Postel. *RFC-791 Internet Protocol*. Information Sciences Institute, USC, CA, September 1981.

[Pos81b] J. Postel, editor. *RFC-793 Transmission Datagram Protocol*. Information Sciences Institute, USC, CA, September 1981.

[Spa89a] Eugene H. Spafford. The Internet Worm: Crisis and Aftermath", journal = "Communications of the ACM. 32(6):678–687, Jun 1989.

[Spa89b] Eugene H. Spafford. The Internet Worm Program: An Analysis. *Computer Communication Review*, 19(1), January 1989.

[Spa90] Eugene H. Spafford. Extending Mutation Testing to Find Environmental Bugs. *Software Practice and Principle*, 20(2):181–189, Feb 1990.

[Sto90] C. Stoll. *The Cuckoo's Egg*. Pocket Books, first edition, 1990.

[Tai89] K. C. Tai. What to do beyond Branch testing. *ACM Software Engineering Notes*, 14(2):58–61, April 1989.

[WC80] J. L. White and E. I. Cohen. A Domain Strategy for Program Testing. *IEEE Transactions on Software Engineering*, SE-5(5):247–257, May 1980.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley Publishing Company, 1995.

[Y^+85] Shen Yu et al. Identifying Error-Prone Software - An Empirical Study. *Transactions on Software Engineering*, SE-11(4):317–323, April 1985.

[ZRT95] G. Ziemba, D. Reed, and P. Traina. *RFC-1858 Security Considerations for IP Fragment Filtering*. Network Working Group, October 1995.