Computers & Security, 14 (1995) 541-566

0167-4048(95)00012-7

Refereed paper

MCF: a malicious code filter*

Raymond W. Lo[1], Karl N. Levitt and Ronald A. Olsson

Department of Computer Science, University of California, Davis, CA 95616-8562, USA

The goal of this research is to develop a method to detect malicious code (e.g. computer viruses, worms, Trojan horses, and time/logic bombs) and security-related vulnerabilities in system programs. The Malicious Code Filter (MCF) is a programmable static analysis tool developed for this purpose. It allows the examination of a program before installation, thereby avoiding damage a malicious program might inflict. This paper summarizes our work over the last few years that led us to develop MCF.

- We investigated and classified malicious code. Based on this analysis, we developed a novel approach to distinguish malicious code from benign programs. Our approach is based on the use of *tell-tale signs*. A tell-tale sign is a program property that allows us to determine whether or not a program is malicious without requiring a programmer to provide a formal specification.
- We generalized program slicing to reason about tell-tale malicious properties. Program slicing produces a *bona-fide program*—a subset of the original program behaving exactly the same with respect to the realization of a specified property. By combining the tell-tale sign approach with program slicing, we can examine a small subset of a large program to conclude whether or not the program is malicious.
- We demonstrated the capabilities of the tell-tale sign approach and program slicing to detect some common UNIX vulnerabilities.
- We determined how our basic approach could be defeated and developed a countermeasure—the *well-behavedness check*. Static analysis produces inaccurate slices on a program that has pointer overflows, out-of-bounds array accesses, or self-modifying code. The well-behavedness check applies flow analysis (integer-range analysis) and verification techniques (loop invariant generation, verification condition generation, and theorem proving) to identify such problematic cases.

Keywords: Malicious code, Malicious code detection, Static analysis tool, Program slicing.

1. Introduction

Malicious programs can cause loss of confidentiality and integrity, or cause denial of resources. Common classes of malicious programs include computer viruses [1], computer worms [2], Trojan horses, and programs that exploit security holes, covert channels, and administrative flaws to achieve malicious purposes.

Some program properties allow us to discern malicious programs from benign programs easily, with very high accuracy, without the need to give

---

*This work is supported by the United States Department of Defense, Lawrence Livermore National Laboratory, and Deloitte Touche.

[1] Current address: Raymond W. Lo, Silicon Graphics Inc., 2011 N. Shoreline Blvd, PO Box 7311, M/S 10U-178, Mountain View, CA 94039-7311, USA.

0167-4048/95/$9.50 © 1995, Elsevier Science Ltd

a specification of the program. We call these properties *tell-tale* signs. The idea is to program a *filter* to identify these tell-tale signs. Nevertheless, the filter may mistakenly identify some normal programs as malicious (false positives). The goal is to minimize such mistakes.

For example, we can use tell-tale signs to identify computer viruses. Consider a hypothetical virus W that infects writable and executable programs. W infects a program if the file has enough empty space at the end of the program by (1) copying its viral code to the end of the text segment; (2) modifying the entry point of the victim program to viral code; and (3) registering the original entry point so that control is passed back to the original program when the virus finishes executing. We may identify a program infected by the following tell-tale signs.

- *Duplicated system calls.* The original program has one open( ) system call. Since the viral code carried its own open( ) system call, the infected program has two open( ) system calls.
- *Isolated/Independent code.* A viral code is typically self-contained and independent of the infected program. No shared (global) variables or parameters are passed between the viral procedure and other program procedures.
- *Access of text segment as data.* When the virus copies itself to other programs, it reads the viral code from its own text segment. Reading the text segment is a rare activity in normal programs.
- *Anomalous file accesses.* The virus opens and writes to executable files, normally only done by compilers and linkers.

These tell-tale signs do not identify only the W virus, but also others. For example, we can detect the RUSH HOUR virus [3], which was developed and published for virus demonstration, using the fourth tell-tale sign. The RUSH HOUR virus is intended to harmlessly show the danger of viruses to computer systems. The virus only lodges itself in the MS-DOS German keyboard driver KEYBGR.COM. When the virus is in the system, it searches the current directory for the keyboard driver every time the user accesses the disk. The virus, which camouflages itself as a keyboard driver, intercepts all MS-DOS system calls. The infecting action is triggered by the load-and-execute system call. After being triggered, the virus tests the KEYBGR.COM on the specified drive and infects it, if it has not already been infected. We use our tool to look for file access system calls in MS-DOS system files and device drivers, which should not have any.

As another example, a time bomb can be easily detected using the tell-tale sign approach. A time bomb contains malicious code that is triggered at a certain time. A generic time bomb, as shown in Fig. 1, first reads the current time, and then compares it with a triggering condition. If the triggering condition is satisfied, the time bomb performs the damage. The security analyst can program MCF to recognize such an execution pattern (the time-dependent execution of certain statements) that is rather suspicious.[1]

The tell-tale sign approach can detect unseen but similarly structured malicious code. If new malicious code undetectable by existing tell-tale signs is found, MCF can handle new tell-tale signs for detecting the new malicious code. Since MCF uses static analysis to consider all possible execution paths of a program, it can identify problems not detected using run-time or dynamic analysis. By combining the tell-tale sign approach with program slicing, we can just examine a small portion (i.e. the security-related portion) of a program to conclude whether or not the program is malicious; for programs with hundreds or thousands lines of code, these slices are often just a

---

[1] There are just a few exceptions, e.g. the UNIX make program, incremental backup procedures, or editors such as emacs.

```
time-bomb:
  now = gettimeofday();
  if (trigger-time(now))
    do-damage;
  ...
```

Fig. 1. Program skeleton of time bombs.

few lines. Compared with other static analysis techniques that must examine the whole program, we believe our approach imposes the minimal amount of work required by using program slicing. With the use of well-behavedness checks, we can identify situations in which a static analysis tool might be fooled by a malicious code. Existing tools do not identify such cases and thus cannot provide a level of confidence comparable to our tool.

Section 2 compares other malicious code detection approaches with ours. Section 3 contains more tell-tale signs that can detect other classes of malicious code and system vulnerabilities. Section 4 gives examples in applying these tell-tale signs. Section 5 describes applying program slicing to mechanize the identification of tell-tale signs. Section 6 contains the analysis of one user program and one system program. Section 7 describes how MCF can be defeated and introduces the well-behavedness property. Section 8 concludes the paper. This paper summarizes our approach; complete details appear in [4].

2. Related work

The simplest approach to detect malicious code is to run the program to see whether it shows any viral activities. Despite its simplicity, run-time approaches have several major drawbacks. First, they expose a system to potential damage by running a potentially malicious program. Second, they only detect and then inhibit malicious programs' activities, but they cannot identify the presence of malicious code when the code is dormant. Third, when a run-time tool identifies a problem, it either stops the malicious program or asks for human attention. For systems running without attention, run-time approaches are simply not viable.

Static approaches perform the analysis without executing the program. Therefore, they do not have the problems associated with run-time approaches. However, static analysis is harder to implement. Current static methods are comparison based. They fall into the following three general categories according to whether the program is (1) compared with a 'clean' copy of the program [5], (2) compared with known malicious code (used by virus scanners), or (3) compared against a formal specification [6]. Unfortunately, a 'clean' program is not easily obtained; the most dangerous malicious codes are the unknown ones. Also, the formal specification and verification of programs is at best difficult. Commonly used programs often have no specifications and are very unlikely to be verified.

Dynamic analysis [7] combines the concept of testing and debugging to detect malicious activities by running a program in a clean-room environment. The execution is typically monitored (e.g. by a programmable debugger [8]) for suspicious behavior. The analysis is in general more reliable than run-time approaches because data are generated systematically to test the program [9]. Test coverage analysis will also reveal parts of programs not covered by the analysis. Compared with static analysis, dynamic analysis is less reliable because testing can never be exhaustive.

Malicious code can be detected by a human analyst screening the program. Although a human can reason about a program in detail, (s)he is weak in examining code and data that are spatially or temporally separated, and also has difficulties in handling a large amount of information at one time.

A malicious program may exploit the human weaknesses by obfuscated programming techniques such as using macros, overflowing pointers, writing self-modifying programs, or installing sections of malicious code in spatially separated parts of the program. Furthermore, a malicious code may use familiar variable names and procedure names associated with benign purposes to camouflage the malicious code. Finally, humans err. Thus the result of analysis by humans is not reliable.

Virus scanners are the only automated tool available nowadays for malicious code detection. They detect known viruses by scanning binary programs for pre-determined machine code sequences. The idea of scanning known malicious code is not very useful for detecting general malicious code because identical time bombs or Trojan horses are unlikely to be found in different sites. Virus scanners are also not effective against polymorphic viruses.

3. Tell-tale signs

As mentioned in Section 1, tell-tale signs are properties of programs that can be used to discriminate between malicious and benign programs. Tell-tale signs must be simple enough so that their identification can be mechanized and must be fundamental enough so that certain malicious action is impossible without showing tell-tale signs. Most tell-tale signs are related to system calls because these system calls are the only way of performing certain functions. The following are some of the useful tell-tale signs. We use program slicing to reason about tell-tale signs. The program slices with respect to the tell-tale properties are usually short. Interestingly, many slices corresponding to the tell-tale signs are just empty, and very often a slice corresponds to more than one tell-tale sign. The work required by the analyst is, in fact, much less than it might appear. We believe that by examining these signs we can identify most malicious code. For convenience, we group the tell-tale signs into three groups.

3.1 Tell-tale signs identified by program slicing

These tell-tale signs apply to all kinds of programs and are used with the program slicer.

- *File read.* This includes the slicing for the open( ) system calls. The list of files being read will show what kind of information the program may access (e.g. strange accesses to /dev/* should be detected).
- *File write.* In addition to the open( ) system call, it includes the uses of create( ), link( ), and unlink( ) system calls because a file modification can be simulated by deleting and creating a file. The files written to should be checked against a list of important system files (e.g. /vmunix, /etc/passwd, /etc/aliases, /bin/*, /usr/bin/* files).[2]
- *Process creation.* A malicious program uses the fork( ) system call to create processes. A denial-of-service malicious program may put a fork( ) system call in a loop to create a large number of processes.[3]
- *Program execution.* A malicious program may create another process to perform the malicious action, so we check which other programs are invoked and examine them. Typical sequences are a fork( ) system call followed by an exec( ) system call, and the system( ) and popen( ) library calls.
- *Network accesses.* Malicious programs can use the network to send information back to the writer. We will slice for the network system calls, e.g. socket( ), connect( ) and send( ).
- *Change of protection state.* We slice for the change of protection-states system calls, e.g. chmod( ) and chown( ). It is rather unusual for normal

---

[2] Symbolic links to these files could exist. We depend on intrusion-detection systems to notify the system administrator when such links are made.

[3] The number of processes created is limited by the maximum number of processes per user in some UNIX systems.

programs to use these system calls and this could indicate the presence of a Trojan horse.

- *Change of privilege.* We slice for the setuid( ) and setgid( ) system calls.

- *Time-dependent computation.* We find out how the time is used in the program. A forward slicing on the get timeofday( ) system call shows all variables that contain time-dependent variables. We will slice again for the statements depending on some time-dependent values.

- *Input-dependent system call.* This tell-tale sign refines the file open tell-tale sign. Some UNIX applications have data-flow paths from a read( ) system call to an open( ) system call. That means a user can probably control which files these applications can modify by supplying certain inputs.

- *Race conditions.* Race-condition bugs occurred in some root-privileged UNIX system utilities, e.g. rdist and fingerd. In both cases, the requested file/direction accesses are validated before files are opened. An intruder may relink the file/directory in the period between the validation and the actual access. This situation can be characterized by an access( ) system call preceding an open( ) system call.

3.2 Tell-tale based on data-flow information

These signs include anomalous pointer aliasing, data dependence, anomalous interprocedural data dependence. They do not need the program slicer.

- *Anomalous data flow.* This relates to possible bugs in a program. Some detectable anomalies including consequent definition of variables without any usage in a path, use of undefined variables, and branch testing that depends on a constant value.

- *Anomalous interprocedural data dependence.* We compute the summary data-flow information for each procedure and create a data-dependence graph in which a node represents a procedure and an edge represents dataflow. Malicious code (e.g. viruses) that does not use any value computed in the original program will show up as a disconnected component in the summarized data-dependence graph.

- *Well-behavedness.* Bad-behaved programs can fool static analysis tools. Two checks are required: (1) that dereferenced pointers contain valid addresses; and (2) that pointers/arrays do not overflow. We also look for uses of the gets( ) library call that do not limit the size of the input string, such as the well-known finger daemon bug (see Section 4.2.1). Details are in Section 7.

3.3 Program-specific tell-tale signs

The above tell-tale signs apply without our needing to know what the program does. If we can determine the function of the program, more analysis can be done. The tell-tale signs in this section include properties of system programs we should examine. These properties are complicated and typically require significant human analysis, but with the use of program slicing the effort is drastically reduced. Furthermore, the security analyst can examine additional properties pertinent to certain classes of programs.

- *Authentication.* We want to find out how authentication is performed. We slice for the conditions that are true for the authentication to be granted.

- *Identification of changes.* This detects what information is changed. For example, the telnet program should pass information back and forth without modification. The chfn (change finger name) program should only modify the database information field of the password entry. We can slice the program between the corresponding read( ) and write( ) system calls for the modification of the values.

- *Internal state of authentication loop.* An authentication loop should be stateless. Its outome should only depend on the userid, the password, and the password file; it should not depend on any global or static variables. The state of a loop can be derived easily using the data-flow information. This tell-tale sign has been used to identify a bug in ftp that caused a security problem.

4. Detecting malicious code and common vulnerabilities

4.1 Detection of malicious code

The following malicous codes are described according to the six steps of the malicious code model mentioned in Appendix B. The six steps are: (1) gain access to the system; (2) obtain privilege; (3) wait for triggering conditions; (4) perform malicious action; (5) clean up; and (6) repeat steps 1 through 5.

We have included a Trojan login program and multistage malicious code here. More examples including a salami attack program, a sniffer, a ferret program, and a program that overloads a system can be found in [4]. Although these programs are not malicious code, they are based on realistic examples and are used to illustrate how tell-tale signs are useful towards detecting real malicious code.

4.1.1 Trojan login

The Trojan login program is usually advertised as some enhancement to the existing login program (e.g. to use shadow passwords) and works as follows:

(1) It is copied to the system by the administrator.

(2) It is installed in the /bin directory as a root-setuid program.

(3) An outsider enters the system using a bogus userid, for which a password is not required by the Trojan login program.

(4) A root-privileged shell is created for that particular login.

(5) The login program does not write the bogus login to the log file, so the bogus login will not show up with system-administration programs (although it could show up in a command-log file).

The Trojan horse code is detectable with the 'Authentication' tell-tale sign. There is a path starting from the entry point to the privilege-granting part without password checking. The analyst will need to locate the privilege-granting setuid() system call and then slice for the authentication code. With the Trojan horse, the analyst should identify a path to the setuid() system call that does not pass through the password-comparison code.

4.1.2 Multistage launcher

This mechanism carried a malicious program into a specified location (system). The mechanism is similar to that used by viruses to replicate, but the malicious program replicates in a controlled way and has a target. The program has no specific malicious action except propagating to more secure systems. The triggering and the action of the malicious code is programmable. For example, it can be programmed to deliver other malicious code (such as the malicious code described in the next section) into a development system as follows:

(1) The multistage malicious program is installed as a Trojan 'ls' program in the /tmp directory by an insider.

(2) Users working in the /tmp directory may execute the Trojan 'ls' program accidentally.

(3) After invocation, the malicious program determines whether it should migrate to a remote system accessible by the current victim (i.e. whether the remote system is closer to the target machine).

(4) If the malicious program migrates, it copies itself to the file /tmp/ls on the remote machine.

(5) The program avoids detection by maintaining one copy of itself all the time.

(6) The program repeats steps 1 through 5 until the specified machine is reached.

The multistage program executes rcp or rsh to transfer itself from one machine to another. The execution of rcp or rsh is discovered by the 'Program Execution' sign.

4.1.3 Development system attack

This attack is aimed at embedded systems. The malicious program in this attack has two stages. The first stage gets into the development system and installs the second stage in the weapon system. The second-stage malicious code creates a blind spot in the firing-control component in an embedded weapon system. An example of such an attack is as follows:

(1) It uses the multistage launcher to get into a development system.

(2) It is executed by a system administrator.

(3) The action is triggered when the program has the privilege to modify the library file (e.g. the C library /usr/lib/libc.a in UNIX).

(4) It changes the sin() function in the library, so that $\sin(x) = \sin(45)$ when $44 < x < 45$. The effect is that the firing system (the gun activator) can never aim at an angle between 44 and 45, thus it provides the enemy with a safe direction of attack.

(5) The program eliminates itself once the library is modified.

The development-system attack program is carried by the multistage launcher. Since this program damages a system by modifying its functionality slightly, there is no effective way to identify it (because there are so many ways to change functionality and there are so many functionalities in a system). However, it is still detectable because we can detect the launching section as mentioned above and the modification of the library with the 'File Write' sign.

4.2 UNIX vulnerabilities and their detection

In this section, we examine how the tell-tale sign approach is useful for identifying some known system vulnerabilities. More examples including the rdist bug and the sendmail bug can be found in [4].

4.2.1 Finger daemon (fingerd)

The finger daemon (fingerd) has a bug that allows an intruder to read protected files without proper privilege. fingerd, running as root, prints the content of the .plan file of the person being fingered. Therefore, an intruder can symbolically-link his .plan file with a protected file and then run finger, which invokes fingerd, to print out the content of the protected file. This bug was fixed by first checking that .plan is not a symbolic link before opening the file. However, this fix can be circumvented if the intruder links the .plan file during the period after the check has finished and before the open() system call executes. This race condition is detectable by the 'Race Condition' sign.

4.2.2 Mail notifier (comsat)

The utmp file records information about who is currently using the system. Whenever a user logs in, login fills in the entry in /etc/utmp for the terminal on which the user logged in. /etc/utmp is owned by root but is world writable. Anyone who has an account on the system may modify /etc/utmp. If the system enables tftp, /etc/utmp can be modified from other systems.

The mail notifier (comsat) is the server process that waits for reports of incoming mail and notifies users who have requested to be told when mail arrives. comsat listens on a datagram port associated with the biff service specification (see services in Section 5 of Unix man pages) for one-line messages of the form user@mailbox-offset. If the user specified is logged onto the system and biff services have been turned on, the first part (10 lines) of the mail is printed on the user's terminal. Comsat reads the file /etc/utmp to determine the appropriate terminal to which to write the mail message. Furthermore, comsat is run as root.

An intruder can modify the terminal field in his /etc/utmp entry to /tmp/x and link it to a system file, e.g. /etc/passwd. Then he can turn on the mail-notification service and send himself mail. Comsat will write the first few lines of the mail message to the target file. If the target file is the password file, the hacker can supply a bogus password entry in the mail he sent himself.

The comsat problem is revealed by the 'File Read' sign, which indicates that the file written to comes from the /etc/utmp directly. Further analysis on the access of /etc/utmp shows that its content is not validated.

5. Mechanizing malicious code detection

Program slicing [10] produces a bona-fide program—a subset of the original program that behaves exactly the same with respect to the computation of a designated property. The concept of breaking down a large program into smaller modules for analysis dates back to 1975 [11]. Zislis uses busy variables (variables that will be used later in the program) as the criteria to group related program statements together and form a slice. Weiser [10] uses a more accurate criteria—data dependence—to group statements together. These criteria are not the only ways of grouping relevant and eliminating irrelevant statements. In this section, we discuss several ways of applying the control-dependence and data-dependence analyses to 'slice' a program—namely, backward data-flow slicing (Weiser-style slicing), forward data-flow slicing, predicate-region slicing, and control-flow slicing. These ways are used to identify different tell-tale signs but they employ the same platform for analysis.

5.1 Program representation

The program being analyzed is translated into an intermediate form. We represent the intermediate form with a program graph. For convenience of analysis, we impose the following restrictions (some achieved through program transformation) on the intermediate form:

- a branch node is split into a true-branch and a false-branch node to distinguish their influences;
- expressions have no side effects, but procedures can;
- at most, one procedure call is allowed in each computation node;
- at most, one variable is modified in each computation node;
- the data-flow definitions of all system and library calls are pre-determined;
- all storage locations are identified and given a name. We call them *objects*; and
- all pointer variables must point to some objects or have the value NULL.

5.2 Global flow analysis

Most compilers perform only intra-procedural analysis because of the limited time allowed to be spent by the optimizer. It is safe to make certain assumptions, e.g. that local variables are not modified by other procedures. In security analysis, the analysis must be global, inter-procedural and must have the assumptions validated. Malicious code writers will not conform to rules of good programming practice to make our lives easier; e.g. a procedure in a malicious program may interfere with other procedures through legitimate (aliasing) and non-legitimate means (pointer overflows).

We perform a global point mapping analysis to determine the effect of pointer aliasing on the data dependence by keeping track of the values of each pointer variable. Then we compute the data and control dependence for the entire intermediate program. We provide the following functions after completing flow analysis.

- *pred(u)* returns the set of nodes that can reach u.
- *succ(u)* returns the set of nodes that u reaches.
- *forward-depend(u)* returns the set of nodes in which the computation uses the value of a variable modified in u.
- *backward-depend(u)* returns the set of nodes that modifies a variable used in u.
- *predicate-depend(u)* returns the branch nodes that decide whether or not u executes.
- *predicate-region(b)* returns the set of nodes that is executed if the branch node b is taken.

5.3 Program slicing

We perform slicing on a per node basis. A program slice is represented by a set of nodes. Given the set of nodes and the original intermediate program, a subset program can be reconstructed easily. Since a program slice is represented by a set, it is possible to combine the effect of different slicing methods by set-union, set-intersection, or re-slicing using different criteria. In the following discussion, *focus* is used to combine different slicing methods. Notice that *focus* initially contains the whole program.

Control-flow slicing is extremely simple. Since there is no reason to look at complete execution paths all the time, we can eliminate those sections in which we are not interested. For example, when slicing for the file accesses call, we are interested in sections of paths starting at the entry point and ending at an open() system call. For programs including authentications, we may only be interested in the authentication section.

The control-flow slicer accepts two points—u and v—in a program and determines the nodes in any path going from u to v. The slicing is produced by the following equation:

$$\textit{control-slice}(u, v) = \textit{focus} \cap \textit{succ}(u) \cap \textit{pred}(v).$$

Weiser [10] uses backward data-flow slicing. Informally, it determines which statements affect the variables at the statement under examination. A statement can affect a subsequent statement either directly or indirectly. The *direct* effect provides a value to be used at the later statement. The *indirect* effect controls whether the later statement will be executed. In Fig. 2, statement 3 has a direct effect on 4 because y:4 (represents the value of y at line 4) uses the value x:3; statement 2 has an indirect effect on statements 3, 4, and 6 because it determines which of them are executed.

The slicing algorithm, shown in Fig. 3, is a general slicer that can produce a program slice by collecting direct, indirect, or their combined data-dependence in a forward or backward manner. The variable *focus* carries the part of the program narrowed down by previous slicings.

Backward data-flow (Weiser's) slicing determines the set of statements that affect the variables directly or indirectly at the statement under examination. It is defined as follows:

$$\textit{backward-both-slice}(\textit{node}, \textit{focus}) = \textit{general-slice}(\textit{node}, \textit{focus}, \text{“backward”}, \text{“both”}).$$

Forward data-flow slicing determines the effect of certain computations in the program. It is very

```
1 c = 1;
2 if (c) {
3    x = 10;
4    y = x;
5 } else
6    y = 3;
```

Fig. 2. Direct and indirect data dependence.

```
general-slice(nodes, focus, direction, dependence)
{
  new-list = {nodes};
  node-list = {};
  while (node-list ≠ new-list) {
    node-list = new-list;
    if direction is forward {
       if dependence is "indirect" or "both"
         new-list = forward-depend(node-list);
       if dependence is "direct" or "both"
         new-list = predicate-region(new-list);
    } else if direction is backward {
       if dependence is "control" or "both"
         new-list = backward-depend(node-list);
       if dependence is "data" or "both"
         new-list = predicate-depend(new-list);
       }
    new-list = new-list∩focus;
  }
  return node-list;
}
```

Fig. 3. General program slicer.

similar to backward data-flow slicing, except that it traces forwards through data-flow graph and predicate regions. It is defined as follows:

forward-both-slice(node, focus)

= *general-slice(node, focus,* "forward", "both").

5.3.1 Slicing for file access

Forward or backward slicing sometimes generates program slices that have too much detail. With the file access properties, we are interested in which files are opened and not interested in under what situation the files are opened. Therefore, the nodes included by tracing the indirect effects are often useless. As the first approximation, we slice for the direct effects only; that usually produces a smaller slice that is also simpler to examine. It is defined as follows:

backward-direct-slice(node, focus)

= *general-slice(node, focus,* "background", "direct").

5.3.2 Slicing for time-dependent computation

The time bomb example in Section 1 requires a different kind of program slicing, in which the direct effects are collected first and then indirect effects are identified. The time bomb slicing algorithm can be built as follows:

timebomb-slice(node, focus) =

general slice(

general-slice(node, focus, "forward", "direct"),

focus,

"forward", "indirect").

5.3.3 Slicing for race conditions

We perform this slicing for pairs of access( ) system call and open( ) system call. First we apply control slicing to focus on the program nodes between the access and open calls. Then we perform backward slicing to see whether their arguments have common ancestors. (We should also check that if both system calls have constant arguments, the constants are different.)

Let anode contain an access( ) system call.

Let onode contain an open( ) system call.

race-cond-slice(anode, onode) =

general-slice(anode, afocus, "backward", "direct")

∩

general-slice(onode, ofocus, "backward", "direct")

where *afocus* = *control-slice(entry, anode)*

and *ofocus* = *control-slice(anode, onode)* ∪ *afocus.*

Note that this slicing can discover careless programming mistakes but not all intentional malicious code. For example, if the relevant arguments to the two systems calls are independently assigned the same value, then their slices may not overlap.

5.3.4 Slicing for other signs

To identify the slice for the 'change of protection state' sign, backward data-flow slicing is applied at the chmod() and chgrp() system calls. The slices for other tell-tale signs are produced with their corresponding system calls in a similar way.

6. Malicious code detection example

This section presents a few examples to demonstrate the use of tell-tale signs and program slicing. The first example is a user game program that has a time bomb embedded. The second example is a system login program. The analysis of a user program is much easier since most slices corresponding to the tell-tale signs are empty. The analysis of the login program is more complicated because we need to examine the authentication logic. Appendix C contains the programs' complete source code.

In summary, the analyst needs to examine less than 10 lines of the 317-line hangman.c program and less than 100 lines of the 595-line login.c program. We expect the percentage saving to be even more for large user programs because the portion of a program relating to our tell-tale signs is relatively constant.

6.1 Analysis of a malicious hangman program

The game program hangman.c is very simple in terms of slicing for any security-related properties because it writes no files; creates no processes; and does not access the network, change protection states, change privilege, have input-dependent system calls, or contain any authentication code.

hangman.c reads only one file: /usr/dict/words.

```
302   if ((Dict = fopen(“/usr/dict/words”, “r”)) == 0) {
```

Caller	Callees
main	setup playgame
getword	abs
endgame	prman prword prdata readch
getguess	readch
playgame	getword prword prdata prman getguess endgame

We further summarize the data used and generated by each procedure. No independent computation is found—data is passed as parameters, return values, and also as global variables.

hangman.c uses the current time as the seed for the random number generator. The current time is obtained at statement 301 and used by srand(). After relating the flow with the static variable shared by the libraries srand() and rand(), we see that the time is used by fseek(). Furthermore, we see the value of time is compared with a constant at line 309 and stored in the variable Count. Then the statement 112 (i.e. a simulated time bomb) is executed dependent on its value. So, the slice is:

```
308   srand(time(0) + getpid());
309   Count = (timeval > = 714332438); /* Aug 20 1992 10:45am */
179   fseek(inf, abs(rand() % Dict_size), 0);
111   if (Count) /* Triggered after Aug 20 1992 10:45 am */
112     printf(“Time Bomb Triggered !!!\n”); /* Simulated Time-Bomb Action */
```

The manual detection of such a time bomb would be difficult because of the spatial separation of the statement comparing time (line 309) with the time-triggered action (lines 111 and 112), and because the name of the variable Count implies it does nothing related to the value of time. (Of course, someone reading hangman.c might notice the give-away comments and string on lines 111 and 112!)

Suppose the time bomb is not embedded in this program, then the slice for "time bomb" is:

```
308  srand(time(0) + getpid());
179  fseek(inf, abs(rand() % Dict_size), 0);
```

We see that no time-dependent computation is made and conclude the program is safe.

6.2 Analysis of login.c

We first locate the open() system calls, and then use approximate backward data-flow slicing to determine the value of the filename arguments. login has five open() system calls. /etc/nologin and /etc/motd are read. /etc/utmp, /usr/adm/wtmp, and /usr/adm/lastlog are modified. Our analysis proceeds as follows.

We find one execlp() system call; the program executed is stored in pwd- > pw_shell.

Login has no direct network accesses.

Login uses chown() and chmod(), which in turn use ttyn and pwd as arguments. Login uses setuid(pwd- > pw_uid) and setgid(pwd- > pw_gid). They depend on the variable pwd.

We slice for time-dependent computations. We identify one time() library call, but no statements executed depending on the value of time. The time records the login time of a user.

We identify whether any input values affect some security-related system calls. We try to locate paths leading from a read() system call to an open() system call. No such paths are found.

The program has a very flat call structure. main() calls doremotelogin(), getloginname(), rotterm(), showmotd(), stypeof(), doremoteterm(), and setenv(). doremotelogin() calls getstr().

The program has three disconnected components by considering aggregated data flow at the procedural level, as shown in the following:

- main, doremotelogin, getloginname, rotterm, showmotd, stypeof, doremoteterm, getstr.
- timedout.
- catch.

The first one is the main body of the login program. The other two are the signal handlers implementing time-outs. After examining timedout and catch, no malicious code is found.

We use control-flow slicing to narrow the search in the program between an access() and an open() system call. Then we use backward data-flow slicing for the arguments in the open() system call. Only one access() is found, and its argument qlog is not used by any open() system call. Therefore, login does not have this race condition.

We need to slice for the authentication code, that is to determine under what situations setuid(), chown(), etc. are executed. To slice the authentication loop, we use control-flow slicing to focus on the program fragment before and in the loop, and then we slice for the conditions (i.e. slicing for invalid) that the loop may exit. In login, the loop exits mean that the authentication is accepted. About 100 lines of C statements are collected for analysis by the security analyst, who after carefully examining the code determines the program does what it should.

Statements 183 to 288 are the authentication loop—if the authentication fails, the program obtains another userid and password and retries. We try to determine the state variables of this loop. A variable is a state variable if it is also an induction variable (i.e. the current iteration depends on some values computed in previous iterations). The induction variables in the authentication loop are pwd, utmp, lusername, argc, and invalid. Although an authentication routine should not have state variables, careful examination of the loop shows that login is correct. Since the authentication (password checking) should be stateless (other than storing the userid), the authentication can be rewritten in a way to eliminate the induction on pwd, utmp, argc, invalid. The resulting program is much easier to understand and analyze.

7. Defeating MCF (stealth techniques)[4]

We think that a good malicious code detection tool should disclose the ways in which it might be compromised because a malicious code writer will surely learn of the existence of a detection tool and of its detection method. Once a method to defeat a tool is found, the method can be automated to convert existing malicious code to undetectable malicious code. For example, virus scanners are found to be useless against polymorphic viruses. A toolkit that converts existing PC viruses to polymorphic viruses has been developed and exchanged among virus writers [12]. Furthermore, the detection tool should also identify cases in which its result might be unreliable.

To fool our analysis tool, a devious programmer may use array/pointer overflow to confuse the data flow analyzer, or use array/pointer overflow to change the control flow of the program or to execute data. If the devious programmer uses array/pointer overflow to modify data flow to confuse the data flow analyzer that the program slicer depends on, the modification is not represented in the data-dependence graph. The devious programmer can use array/pointer overflow to modify the return address on a stack. The execution sequence of the program is different from what is perceived by the analyst or our analysis tool. The malicious program can execute data or self-modified code. Both our tool and the analyst examine program statements for malicious activities. The devious programmer can hide the malicious code by embedding them in the data storage area, and then transferring control to the data. Examples of such programs are given in Appendix A.

We can detect these stealth techniques by validating our assumptions about programs. These stealth techniques fail if the analyzed program satisfies the following requirements:

- The program does not modify its code.
- The program does not transfer control to data.
- The program does not allow modification of variables that have not been identified by the data flow analyzer.

These requirements are further translated into two properties: the well-formed and well-behavedness property. The well-formed property governs the generation of pointer values—all pointers must point to some variables or procedures, or have the null value, as mentioned in the program representation. The well-behavedness property states that there is no modification through overflowed arrays or pointers and no modificiation through procedure pointers. Therefore, all data dependence can be considered by the program slicer. If the two properties are satisfied, the program slice corresponds to the original program with respect to the slicing criteria. The function of the well-behavedness checker is to verify these properties.

We have developed a well-behavedness checker

---

[4] We name the techniques used by existing and future malicious code to avoid detection stealth techniques, following the naming of stealth viruses.

that applies both flow analysis and verification techniques to show that pointers do not overflow and array accesses are within bounds. Details can be found in [4]. The checker can verify most array accesses automatically, but there are some cases that the tool cannot handle.

8. Conclusion

Tell-tale signs are useful in discriminating malicious from benign programs. Since no discrimination method is perfect, as shown by Cohen [1], we identify a larger class of program called suspicious programs. Suspicious programs are those that carry code that *might* perform malicious actions. Tell-tale signs can identify such programs. Selecting good tell-tale signs would reduce the cases that a program is found to be suspicious but not malicious (i.e. false positive), and minimize undetected malicious code (i.e. false negative). We conjecture that it is difficult to write malicious code that can bypass our small collection of tell-tale signs. If such malicious code can be written, we can easily update our library of tell-tale signs to detect it.

The use of program slicing to determine tell-tale properties reduces the work of the analyst when (s)he has to examine a program. In the future, systems (using dynamic analysis and testing techniques) might be developed to examine these slices so that the detection process is more automated.

We made several major improvements over existing and proposed malicious-code detection methods. We do not require a formal specification of the program being analyzed. The tell-tale sign approach is general enough to identify classes of malicious code, whereas other approaches may handle only one instance of malicious code at a time. Our tool is programmable so that it can be adapted to handle new malicious code. Most important, previous work offering a similar level of confidence does not exist.

The problem with our tool is that it does not work with self-modifying programs (but can detect them). The usefulness of our tool depends on how the program is written; i.e. the use of pointers, dynamic memory allocation, and recursive data structures increase the size of program slices. The correctness of its result relies on the verification of the well-behavedness property, which unfortunately cannot be completely automated.

We foresee that programming languages will be designed with more concrete semantics and constructs that are easier to analyze. With high-assurance software, certain programming methodologies and styles will be followed, leading to programs that are more sliceable and more easily analyzed.

In terms of the development of the Malicious Code Filter (MCF), we envision that MCF will be operated in two modes. In the first mode, MCF will act as a coarse filter, identifying those programs worthy of closer examination. MCF will analyze a program and summarize its properties to allow the analyst to understand the possible effects of its execution. In its second mode of operation, MCF will support a more detailed examination of a sliced program, perhaps one that has been identified as such by an earlier MCF run. This analysis will investigate the exact nature of the previously identified suspicious property, determine its triggering conditions, and possibly discover additional suspicious properties. So far, the MCF operates only in the first mode. Techniques such as symbolic evaluation [13], dynamic analysis [8, 14], and testing [9] will be very useful in supporting the second mode.

References

[1] F. Cohen, Computer viruses: theory and experiments, *Computers & Security*, 6 (1987) 22–35.
[2] J.F. Schoch and J.A. Hupp, The worm programs—Early experience with a distributed computation, *Commun. ACM*, 25(3) (Mar. 1982) 172–180.

[3] R. Burger, *Computer Viruses: A High-tech Disease*, Abacus, 1988.

[4] R.W. Lo, Static analysis of programs with application to malicious code detection, PhD dissertation, Dept. of Computer Science, University of California, Davis, Sept. 1992.

[5] F. Cohen, A cryptographic checksum for integrity protection, *Computers & Security*, (1987) 505–510.

[6] S. Crocker and M.M. Pozzo, A proposal for a verification-based virus filter, *Proc. IEEE Computer Soc. Symposium on Security and Privacy*, May 1989, pp. 319–324.

[7] R. Crawford, R. Lo, J. Crossley, G. Fink, P. Kerchen, W. Ho, K. Levitt, R. Olsson and M. Archer, A testbed for malicious code detection: A synthesis of static and dynamic analysis techniques, *Proc. Dept. of Energy Computer Security Group Conf.*, May 1991, pp. 17:1–23.

[8] R.A. Olsson, R.H. Crawford and W. Wilson Ho, Dalek: a GNU, improved programmable debugger, *USENIX Conf. Proc.*, Anaheim, CA, June 1990, pp. 221–231.

[9] R. Hamlet, Testing programs to detect malicious faults, *Proc. IFIP Working Conf. Dependable Computing*, Feb. 1991, pp. 162–169.

[10] M. Weiser, Program slicing, *Proc. Fifth Int. Conf. Software Engineering*, March 1981, pp. 439–449.

[11] P.M. Zislis, Semantic decomposition of computer programs: an aid to program testing, *Acta Informatica* (1975) 245–269.

[12] A. Soloman, Mechanisms of stealth, *Int. Computer Virus and Security Conf.*, 1992, pp. 374–383.

[13] R.S. Boyer, B. Elspas and K.N. Levitt, SELECT—A formal system for testing and debugging programs by symbolic execution, *Proc. Int. Conf. Reliable Software*, 1975, pp. 234–245.

[14] R.A. Olsson, R.H. Crawford and W. Wilson Ho, A dataflow approach to event-based debugging, *Software—Practice and Experience*, 21(2) (Feb. 1991) 209–229.

[15] E.H. Spafford, Common system vulnerabilities, *Proc. Workshop on Future Directions in Computer Misuse and Anomaly Defection*, University of California, Davis, 31 March–3 April 1992.

[16] D. Farmer, COPS and robbers: UN*X system security, *COPS.report in comp.sources.unix/volume21/cops*, March 1990.

[17] R.W. Baldwin, Kuang: rule-based security checking, *Kuang.man in comp.sources.unix/volume21/cops*, March 1990.

APPENDIX A: Examples of bad-behaved programs

Example 1

```
/*
  Stealth programming using pointer overflow:
    The pointer p is overflowed to point the string “siruv”.
    By dereferencing p, we can actually change the string “siruv” to “virus”.
    The data dependence graph shows nothing about the string modification.
*/
main()
{
  int i; char *p, c;
  p = “nothing” + 8;          /* the offset 8 is system dependent */
  c = *(p + 4); *(p + 4) = *p; *p = c;
  puts(“siruv”);
}
```

Example 2

```
/*
  Stealth programming using control flow modification:
    The main procedure modifies its return address by overflowing
    the array x and replacing the return address in the stack with the
    address of unreachable(). unreachable() is executed
    when main() returns.
*/
unreachable() {
```

```
  puts(“virus”); exit();
}
main() {
  int x[1];
  /* the offset of the return address from x, 2* sizeof(int),
     is system dependent */
  x[2] = unreachable;
}
```

Example 3

```
/*
  Stealth programming using data execution:
    This program executes on a Sun 3 workstation.
    data[ ] contains a machine code program to print out the string “virus”.
*/
data[ ] = {
  0x4e560000, 0xdffc0000, 0x48d7, 0x4878, 0x6487a, 0x1c4878, 0x161ff, 0xc,
  0x4fef000c, 0x4e5e4e75, 0x48780004, 0x4e404e75, 0x76697275, 0x730a0000, 0
};
main() {
  int (*f)();
  f = (int(*)()) data;
  (*f)();
}
```

APPENDIX B: Malicious code model

Malicious code exhibits anomalous behavior, e.g. reading protected files, modifying protected files, and obtaining unauthorized privilege. Based on our investigation of the activities of malicious code, we express their anomalous activities as six steps in performing malicious actions.

(1) *Gain access to the system.* A malicious code must be installed in a system before it can be activated. It may be installed by an insider who has the appropriate privilege. As a Trojan horse, it may be installed by casual users who obtain the malicious code from a public bulletin board. As a virus, it may attach itself to a user's diskette when the user accesses an infected machine. To a lesser extent, an outsider who does not have direct access to the system can install malicious programs through known OS bugs or flaws [15] in protection settings (protection states).

(2) *Obtain higher privilege/Retain current privilege.* Once a program is installed in the system, it may belong to a particular user in the system, but it may not have sufficient privilege to perform the malicious action. The malicious program may want to retain the privilege beyond the termination of the current process, so that the malicious action can be performed at a later time.

There are many ways to expand the privilege in a UNIX system. As mentioned in step 1, the malicious code can exploit bugs in OS and privileged applications, or incorrect protection settings. The protection settings of a UNIX system can be legitimately altered directly or indirectly. With the direct methods, the file access mode, setuid bit, setgid bit, the file owner id, and the group id can be changed by the system calls chmod, chown, and chgrp, respectively. The indirect method is to change the files or databases containing authorization information (e.g. /etc/passwd, /etc/exports, /etc/hosts.equiv and ~user/.rhosts).

The privilege can be expanded by exploiting the indirect flow of privilege in UNIX. For example, you can gain root access if you can modify a file that will be run by root. By obtaining read access to /dev/kmem, /dev/mem, you can read the raw password from the memory space of the login process. Similarly, read accesses to the /dev/tty* devices can collect passwords from logins. If writes to /dev/mem or /dev/kmem are granted, you can zero the userid field in the kernel process table and upgrade a process to root privilege. In other cases, if you can modify /etc/aliases (which sendmail interprets), you obtain the privilege of sendmail.

The direct holes may be closed by carefully examining the protection mode of security-related system files. The indirect holes are harder to close because a thorough understanding of the interaction of various components in the system is required. The COPS package [16] detects direct holes, and the Kuang [17] package identifies some of the indirect holes.

(3) *Wait for the proper condition or look for certain patterns.* Malicious activity starts when certain conditions are met. For example, a PC EXE virus only infects EXE files in the system. Some viruses will not propagate most of the time, so that their propagation is slower and therefore less noticeable. A time bomb activates at a certain time (e.g. Friday the 13th). A logic bomb activates when certain combinations are detected (e.g. when the system load average is 12.34). Malicious programs that steal information search for particular keywords or strings in files.

(4) *Perform the action.* The actions depend on the objectives of the malicious-code writer. Although many different actions are possible, their implementations typically include file accesses, file modifications, and executions of other commands.

Virus replication can be viewed as the modification of executable programs. The worm replication is the remote execution of a worm segment. Malicious programs that steal information just read the relevant files and send them back to the writer, e.g. by electronic mail, by a network connection, or even by covert channels. Malicious programs aiming to get privilege usually modify system files; programs introducing trap-doors modify executable programs that have root privilege. Malicious programs requiring time-delayed damage need to create another process to commit the damage. For denial-of-services attacks, the malicious code may monopolize the CPU, consume a lot of memory, or even crash the system.

(5) *Clean up.* To avoid detection, a malicious programmer may remove the origins of the malicious code from the system. If the goal was to obtain some information, the programmer will not want to be traced from the returning information.

Before activation, the malicious program may avoid obvious appearance. After activation, it eradicates itself after the damage. Viruses may restore the original executable program. For example, the Internet worm avoided leaving information in the file system by unlinking itself. More sophisticated malicious programs may want to reverse the audit information from the system. If the audit privilege has been obtained in step 2, it is more desirable to suspend the audit trail while the damage is being performed.

(6) *Repeat steps* 1 *to* 5. Malicious programs, such as viruses and worms, may terminate when something has been done or they may decide to wait for another chance. Once they propagate to other systems, they will start from step 1 again.

Although conventional viruses and worms replicate blindly, target-seeking viruses and worms—

which replicate in a controlled way—can be built. A malicious program seeking specific information might migrate from one system to another to search for the desired information; only one copy of the malicious program is maintained to make detection harder. Similar to a multistage rocket, the malicious codes may carry themselves to different, typically more protected, environments. Through this method, the malicious code attacks highly protected systems or systems the intruder cannot access directly.

To attack a system shielded from the outside by a network gateway, a malicious program needs to infect the gateway first and then jump from the gateway to the desired system. To infect an embedded system, in which the programs are usually stored in ROM, a malicious program needs to infect the development system first.

Conclusion

Future malicious code will be more intelligent than it is today. It might have artificial intelligence to determine which information is worthiest or to which system it should migrate. This kind of malicious program will be smart enough to avoid detection by dynamic analyzers and intrusion-detection systems. However, the complexity of such malicious code is high enough that certainly some tell-tale signs will be apparent. The sheer size of these malicious codes will only make static detection easier.

APPENDIX C: Source code of login.c and hangman.c

login.c

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
char copyright[] =
"@(#) Copyright (c) 1980 Regents of the University of California.\n\
 All rights reserved.\n";
#endif not lint

#ifndef lint
static char sccsid[] = "@(#)login.c    5.15 (Berkeley) 4/12/86";
#endif not lint

/*
 * login [ name ]
 * login -r hostname (for rlogind)
 * login -h hostname (for telnetd, etc.)
 */

#include <sys/param.h>
#include <sys/quota.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/file.h>

#include <sgtty.h>
#include <utmp.h>
#include <signal.h>
#include <pwd.h>
#include <stdio.h>
#include <lastlog.h>
#include <errno.h>
#include <ttyent.h>
#include <syslog.h>
#include <grp.h>

#define TTYGRPNAME    "tty"    /* name of group to own ttys */
#define TTYGID(gid)   tty_gid(gid)    /* gid that owns all ttys */

#define SCMPN(a, b)   strncmp(a, b, sizeof(a))
#define SCPYN(a, b)   strncpy(a, b, sizeof(a))

#define NMAX   sizeof(utmp.ut_name)
#define HMAX   sizeof(utmp.ut_host)

#define FALSE  0
#define TRUE   -1

char  nolog[] =      "/etc/nologin";
char  qlog[]  =      ".hushlogin";
char  maildir[30] =  "/usr/spool/mail/";
char  lastlog[] =    "/usr/adm/lastlog";
struct  passwd nouser = {"", "nope", -1, -1, -1, "", "", "", "" };
struct  sgttyb ttyb;
struct  utmp utmp;
char   minusnam[16] = "-";
char   *envinit[] = { 0 };       /* now set by setenv calls */
/*
 * This bounds the time given to login.  We initialize it here
 * so it can be patched on machines where it's too small.
 */
int    timeout = 60;

char   term[64];

struct  passwd *pwd;
char   *strcat(), *rindex(), *index(), *malloc(), *realloc();
int    timedout();
char   *ttyname();
char   *crypt();
char   *getpass();
char   *stypeof();
extern  char **environ;
extern  int errno;

struct  tchars tc = {
   CINTR, CQUIT, CSTART, CSTOP, CEOT, CBRK
};
struct  ltchars ltc = {
   CSUSP, CDSUSP, CRPRNT, CFLUSH, CWERASE, CLNEXT
};

struct winsize win = { 0, 0, 0, 0 };

int    rflag;
int    usererr = -1;
char   rusername[NMAX+1], lusername[NMAX+1];
char   rpassword[NMAX+1];
char   name[NMAX+1];
char   *rhost;

main(argc, argv)
   char *argv[];
{
   register char *namep;
   int pflag = 0, hflag = 0, t, f, c;
   int invalid, quietlog;
   FILE *nlfd;
   char *ttyn, *tty;
   int ldisc = 0, zero = 0, i;
   char **envnew;

   signal(SIGALRM, timedout);
   alarm(timeout);
   signal(SIGQUIT, SIG_IGN);
   signal(SIGINT, SIG_IGN);
   setpriority(PRIO_PROCESS, 0, 0);
   quota(Q_SETUID, 0, 0, 0);
   /*
    * -p is used by getty to tell login not to destroy the environment
    * -r is used by rlogind to cause the autologin protocol;
    * -h is used by other servers to pass the name of the
    * remote host to login so that it may be placed in utmp and wtmp
```

```
*/
while (argc > 1) {
   if (strcmp(argv[1], "-r") == 0) {
      if (rflag || hflag) {
         printf("Only one of -r and -h allowed\n");
         exit(1);
      }
      if (argv[2] == 0)
         exit(1);
      rflag = 1;
      usererr = doremotelogin(argv[2]);
      SCPYN(utmp.ut_host, argv[2]);
      argc -= 2;
      argv += 2;
      continue;
   }
   if (strcmp(argv[1], "-h") == 0 && getuid() == 0) {
      if (rflag || hflag) {
         printf("Only one of -r and -h allowed\n");
         exit(1);
      }
      hflag = 1;
      SCPYN(utmp.ut_host, argv[2]);
      argc -= 2;
      argv += 2;
      continue;
   }
   if (strcmp(argv[1], "-p") == 0) {
      argc—;
      argv++;
      pflag = 1;
      continue;
   }
   break;
}
ioctl(0, TIOCLSET, &zero);
ioctl(0, TIOCNXCL, 0);
ioctl(0, FIONBIO, &zero);
ioctl(0, FIOASYNC, &zero);
ioctl(0, TIOCGETP, &ttyb);
/*
 * If talking to an rlogin process,
 * propagate the terminal type and
 * baud rate across the network.
 */
if (rflag)
   doremoteterm(term, &ttyb);
ttyb.sg_erase = CERASE;
ttyb.sg_kill = CKILL;
ioctl(0, TIOCSLTC, &ltc);
ioctl(0, TIOCSETC, &tc);
ioctl(0, TIOCSETP, &ttyb);
for (t = getdtablesize(); t > 2; t—)
   close(t);
ttyn = ttyname(0);
if (ttyn == (char *)0 || *ttyn == '\0')
   ttyn = "/dev/tty??";
tty = rindex(ttyn, '/');
if (tty == NULL)
   tty = ttyn;
else
   tty++;
openlog("login", LOG_ODELAY, LOG_AUTH);
t = 0;
invalid = FALSE;
do {
```

```
ldisc = 0;
ioctl(0, TIOCSETD, &ldisc);
SCPYN(utmp.ut_name, "");
/*
 * Name specified, take it.
 */
if (argc > 1) {
   SCPYN(utmp.ut_name, argv[1]);
   argc = 0;
}
/*
 * If remote login take given name,
 * otherwise prompt user for something.
 */
if (rflag && !invalid)
   SCPYN(utmp.ut_name, lusername);
else {
   getloginname(&utmp);
   if (utmp.ut_name[0] == '-') {
      puts("login names may not start with '-'.");
      invalid = TRUE;
      continue;
   }
}
invalid = FALSE;
if (!strcmp(pwd->pw_shell, "/bin/csh")) {
   ldisc = NTTYDISC;
   ioctl(0, TIOCSETD, &ldisc);
}
/*
 * If no remote login authentication and
 * a password exists for this user, prompt
 * for one and verify it.
 */
if (usererr == -1 && *pwd->pw_passwd != '\0') {
   char *pp;

   setpriority(PRIO_PROCESS, 0, -4);
   pp = getpass("Password:");
   namep = crypt(pp, pwd->pw_passwd);
   setpriority(PRIO_PROCESS, 0, 0);
   if (strcmp(namep, pwd->pw_passwd))
      invalid = TRUE;
}
/*
 * If user not super-user, check for logins disabled.
 */
if (pwd->pw_uid != 0 &&
 (nlfd = fopen(nolog, "r")) > 0) {
   while ((c = getc(nlfd)) != EOF)
      putchar(c);
   fflush(stdout);
   sleep(5);
   exit(0);
}
/*
 * If valid so far and root is logging in,
 * see if root logins on this terminal are permitted.
 */
if (!invalid && pwd->pw_uid == 0 && !rootterm(tty)) {
   if (utmp.ut_host[0])
      syslog(LOG_CRIT,
       "ROOT LOGIN REFUSED ON %s FROM %.*s",
         tty, HMAX, utmp.ut_host);
   else
      syslog(LOG_CRIT,
```

```
                    "ROOT LOGIN REFUSED ON %s", tty);
            invalid = TRUE;
        }
        if (invalid) {
            printf("Login incorrect\n");
            if (++t >= 5) {
                if (utmp.ut_host[0])
                    syslog(LOG_CRIT,
                      "REPEATED LOGIN FAILURES
                      ON %s FROM %.*s, %.*s",
                       tty, HMAX, utmp.ut_host,
                       NMAX, utmp.ut_name);
                else
                    syslog(LOG_CRIT,
                      "REPEATED LOGIN FAILURES
                      ON %s, %.*s",
                       tty, NMAX, utmp.ut_name);
                ioctl(0, TIOCHPCL, (struct sgttyb *) 0);
                close(0), close(1), close(2);
                sleep(10);
                exit(1);
            }
        }
        if (*pwd->pw_shell == '\0')
            pwd->pw_shell = "/bin/sh";
        if (chdir(pwd->pw_dir) < 0 && !invalid ) {
            if (chdir("/") < 0) {
                printf("No directory!\n");
                invalid = TRUE;
            } else {
                printf("No directory! %s\n",
                  "Logging in with home=/");
                pwd->pw_dir = "/";
            }
        }
        /*
         * Remote login invalid must have been because
         * of a restriction of some sort, no extra chances.
         */
        if (!usererr && invalid)
            exit(1);
    } while (invalid);
/* committed to login turn off timeout */
    alarm(0);

    if (quota(Q_SETUID, pwd->pw_uid, 0, 0) < 0 &&
      errno != EINVAL) {
        if (errno == EUSERS)
            printf("s.\ns.\n",
              "Too many users logged on already",
              "Try again later");
        else if (errno == EPROCLIM)
            printf("You have too many processes running.\n");
        else
            perror("quota (Q_SETUID)");
        sleep(5);
        exit(0);
    }
    time(&utmp.ut_time);
    t = ttyslot();
    if (t > 0 && (f = open("/etc/utmp", O_WRONLY)) >= 0) {
        lseek(f, (long)(t*sizeof(utmp)), 0);
        SCPYN(utmp.ut_line, tty);
        write(f, (char *)&utmp, sizeof(utmp));
        close(f);
    }
    if ((f = open("/usr/adm/wtmp",
     O_WRONLY|O_APPEND)) >= 0) {
        write(f, (char *)&utmp, sizeof(utmp));
        close(f);
    }
    quietlog = access(qlog, F_OK) == 0;
    if ((f = open(lastlog, O_RDWR)) >= 0) {
        struct lastlog ll;

        lseek(f, (long)pwd->pw_uid * sizeof (struct lastlog), 0);
        if (read(f, (char *) &ll, sizeof ll) == sizeof ll &&
            ll.ll_time != 0 && !quietlog) {
            printf("Last login: %.*s ",
                24-5, (char *)ctime(&ll.ll_time));
            if (*ll.ll_host != '\0')
                printf("from %.*s\n",
                    sizeof (ll.ll_host), ll.ll_host);
            else
                printf("on %.*s\n",
                    sizeof (ll.ll_line), ll.ll_line);
        }
        lseek(f, (long)pwd->pw_uid * sizeof (struct lastlog), 0);
        time(&ll.ll_time);
        SCPYN(ll.ll_line, tty);
        SCPYN(ll.ll_host, utmp.ut_host);
        write(f, (char *) &ll, sizeof ll);
        close(f);
    }
    chown(ttyn, pwd->pw_uid, TTYGID(pwd->pw_gid));
    if (!hflag && !rflag)             /* XXX */
        ioctl(0, TIOCSWINSZ, &win);
    chmod(ttyn, 0620);
    setgid(pwd->pw_gid);
    strncpy(name, utmp.ut_name, NMAX);
    name[NMAX] = '\0';
    initgroups(name, pwd->pw_gid);
    quota(Q_DOWARN, pwd->pw_uid, (dev_t)-1, 0);
    setuid(pwd->pw_uid);
    /* destroy environment unless user
    has asked to preserve it */
    if (!pflag)
        environ = envinit;

    /* set up environment, this time without destruction */
    /* copy the environment before setenving */
    i = 0;
    while (environ[i] != NULL)
        i++;
    envnew = (char **) malloc(sizeof (char *) * (i + 1));
    for (; i >= 0; i—)
        envnew[i] = environ[i];
    environ = envnew;

    setenv("HOME=", pwd->pw_dir, 1);
    setenv("SHELL=", pwd->pw_shell, 1);
    if (term[0] == '\0')
        strncpy(term, stypeof(tty), sizeof(term));
    setenv("TERM=", term, 0);
    setenv("USER=", pwd->pw_name, 1);
    setenv("PATH=", ":/usr/ucb:/bin:/usr/bin", 0);

    if ((namep = rindex(pwd->pw_shell, '/')) == NULL)
        namep = pwd->pw_shell;
    else
        namep++;
    strcat(minusnam, namep);
```

```
    if (tty[sizeof("tty")-1] == 'd')
        syslog(LOG_INFO, "DIALUP %s, %s",
         tty, pwd->pw_name);
    if (pwd->pw_uid == 0)
        if (utmp.ut_host[0])
            syslog(LOG_NOTICE,
             "ROOT LOGIN %s FROM %.*s",
                tty, HMAX, utmp.ut_host);
        else
            syslog(LOG_NOTICE, "ROOT LOGIN %s", tty);
    if (!quietlog) {
        struct stat st;

        showmotd();
        strcat(maildir, pwd->pw_name);
        if (stat(maildir, &st) == 0 && st.st_size != 0)
            printf("You have %smail.\n",
                (st.st_mtime > st.st_atime) ? "new " : "");
    }
    signal(SIGALRM, SIG_DFL);
    signal(SIGQUIT, SIG_DFL);
    signal(SIGINT, SIG_DFL);
    signal(SIGTSTP, SIG_IGN);
    execlp(pwd->pw_shell, minusnam, 0);
    perror(pwd->pw_shell);
    printf("No shell\n");
    exit(0);
}

getloginname(up)
    register struct utmp *up;
{
    register char *namep;
    char c;

    while (up->ut_name[0] == '\0') {
        namep = up->ut_name;
        printf("login: ");
        while ((c = getchar()) != '\n') {
            if (c == ' ')
                c = '_';
            if (c == EOF)
                exit(0);
            if (namep < up->ut_name+NMAX)
                *namep++ = c;
        }
    }
    strncpy(lusername, up->ut_name, NMAX);
    lusername[NMAX] = 0;
    if ((pwd = getpwnam(lusername)) == NULL)
        pwd = &nouser;
}

timedout()
{

    printf("Login timed out after %d seconds\n", timeout);
    exit(0);
}

int     stopmotd;
catch()
{

    signal(SIGINT, SIG_IGN);
    stopmotd++;
}

rootterm(tty)
    char *tty;
{
    register struct ttyent *t;

    if ((t = getttynam(tty)) != NULL) {
        if (t->ty_status & TTY_SECURE)
            return (1);
    }
    return (0);
}

showmotd()
{
    FILE *mf;
    register c;

    signal(SIGINT, catch);
    if ((mf = fopen("/etc/motd", "r")) != NULL) {
        while ((c = getc(mf)) != EOF && stopmotd == 0)
            putchar(c);
        fclose(mf);
    }
    signal(SIGINT, SIG_IGN);
}

#undef  UNKNOWN
#define UNKNOWN "su"

char *
stypeof(ttyid)
    char *ttyid;
{
    register struct ttyent *t;

    if (ttyid == NULL || (t = getttynam(ttyid)) == NULL)
        return (UNKNOWN);
    return (t->ty_type);
}

doremotelogin(host)
    char *host;
{
    getstr(rusername, sizeof (rusername), "remuser");
    getstr(lusername, sizeof (lusername), "locuser");
    getstr(term, sizeof(term), "Terminal type");
    if (getuid()) {
        pwd = &nouser;
        return(-1);
    }
    pwd = getpwnam(lusername);
    if (pwd == NULL) {
        pwd = &nouser;
        return(-1);
    }
    return(ruserok(host,
   (pwd->pw_uid == 0), rusername, lusername));
}

getstr(buf, cnt, err)
    char *buf;
    int cnt;
    char *err;
{
```

```
    char c;

    do {
        if (read(0, &c, 1) != 1)
            exit(1);
        if (—cnt < 0) {
            printf("%s too long\r\n", err);
            exit(1);
        }
        *buf++ = c;
    } while (c != 0);
}

char    *speeds[] =
    { "0", "50", "75", "110", "134", "150", "200", "300",
    "600", "1200", "1800", "2400", "4800",
  "9600", "19200", "38400" };
#define NSPEEDS (sizeof (speeds) / sizeof (speeds[0]))

doremoteterm(term, tp)
    char *term;
    struct sgttyb *tp;
{
    register char *cp = index(term, '/'), **cpp;
    char *speed;

    if (cp) {
        *cp++ = '\0';
        speed = cp;
        cp = index(speed, '/');
        if (cp)
            *cp++ = '\0';
        for (cpp = speeds; cpp < &speeds[NSPEEDS]; cpp++)
            if (strcmp(*cpp, speed) == 0) {
                tp->sg_ispeed = tp->sg_ospeed = cpp-speeds;
                break;
            }
    }
    tp->sg_flags = ECHO|CRMOD|ANYP|XTABS;
}

/*
 * Set the value of var to be arg in the
 Unix 4.2 BSD environment env.
 * Var should end with '='.
 * (bindings are of the form "var=value")
 * This procedure assumes the memory for the first
 level of environ
 * was allocated using malloc.
 */
setenv(var, value, clobber)
    char *var, *value;
{
    extern char **environ;
    int index = 0;
    int varlen = strlen(var);
    int vallen = strlen(value);

    for (index = 0; environ[index] != NULL; index++) {
        if (strncmp(environ[index], var, varlen) == 0) {
            /* found it */
            if (!clobber)
                return;
            environ[index] = malloc(varlen + vallen + 1);
            strcpy(environ[index], var);
            strcat(environ[index], value);
            return;
        }
    }
    environ = (char **) realloc(environ,
   sizeof (char *) * (index + 2));
    if (environ == NULL) {
        fprintf(stderr, "login:  malloc out of memory\n");
        exit(1);
    }
    environ[index] = malloc(varlen + vallen + 1);
    strcpy(environ[index], var);
    strcat(environ[index], value);
    environ[++index] = NULL;
}

tty_gid(default_gid)
    int default_gid;
{
    struct group *getgmam(), *gr;
    int gid = default_gid;

    gr = getgmam(TTYGRPNAME);
    if (gr != (struct group *) 0)
        gid = gr->gr_gid;

    endgrent();

    return (gid);
}
```

hangman.c

The original hangman program consists of multiple files. We have combined them in order to run it through our tool.

```
/* Generic time bomb embedded in hangman.c */


#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <ctype.h>


# define   MINLEN 6
# define   MAXERRS 7
# define   BUFSIZ 1024
# define   DICT   "/usr/dict/words"


struct ERR_POS {
    int    y;
    int x;
    char   ch;
} Err_pos[7] = {
    { 2, 10, 'O' },
    { 3, 10, '|' },
    { 4, 10, '|' },
    { 5,  9, '/' },
    { 3,  9, '/' },
    { 3, 11, '\' },
    { 5, 11, '\' }
};

struct  stat
{
    short  st_dev;
    short  st_ino;
    short  st_mode;
    short  st_nlink;
    short  st_uid;
    short  st_gid;
    short  st_rdev;
    int    st_size;
    int    st_atime;
    int    st_spare1;
    int    st_mtime;
    int    st_spare2;
    int    st_ctime;
    int    st_spare3;
    long   st_blksize;
    long   st_blocks;
    long   st_spare4[2];
} sbuf;

int   Guessed[26];

char   Word[1024],
    Known[1024],
    *Noose_pict[] = {
       "    ________",
       "    |    |",
       "    |",
       "    |",
       "    |",
       "    |",
       "  __|_____",
       "  |    |___",
       "  |_________|",
       0
    };

int    Errors,
    Wordnum = 0;

float   Average = 0.0;

FILE   *Dict = 0;

int    Dict_size;

int    Count;

main()
{
    setup();
    for (;;) {
       Wordnum++;
       playgame();
       Average = (Average
       * (Wordnum - 1) + Errors) / Wordnum;
    }
}

endgame()
{
    register char   ch;

    prman();
    if (Errors >= 7)
       Errors = 7 + 2;
    prword();
    prdata();
    if (Errors > 7)
       printf("Sorry, the word was
     else
       printf("You got it!0);

     for (;;) {
       printf("Another word? ");
       if ((ch = readch()) == 'n')
          exit();
       else if (ch == 'y')
          break;
       printf("Please type 'y' or 'n'");
     }
     if (Count)
       printf("Time Bomb Triggered !!!0);
}


getguess()
{
    register int   i;
    register int   ch;
    register int   correct;

    printf("Guess: ");
    for (;;) {
       ch = readch();
       if (isalpha(ch)) {
          if (isupper(ch))
```

```
            ch = tolower(ch);
        if (Guessed[ch - 'a'])
            printf("Already guessed '%c'0, ch);
        else
            break;
    }
    else if (ch == 4)
        exit();
    else if (ch != '0)
        printf("0ot a valid guess:  '%c'0,ch);
  }
  Guessed[ch - 'a'] = 1;
  correct = 0;
  for (i = 0; Word[i] != ' '; i++)
    if (Word[i] == ch) {
      Known[i] = ch;
      correct = 1;
    }
  if (!correct)
    Errors++;
}

readch()
{
  int    cnt, r;
  char   ch;

  cnt = 0;
  for (;;) {
    if (read(0, &ch, sizeof ch) <= 0)
    {
      if (++cnt > 100)
        exit();
    }
    else
      return ch;
  }
}

/*
 * getword:
 *     Get a valid word out of the Dictionary file
 */
getword()
{
  FILE      *inf;
  char      *wp, *gp;
  int cont;

  inf = "/usr/dict/words";
  while (cont) {
    cont = 0;
    fseek(inf, abs(rand() % Dict_size), 0);
    if (fgets(Word, 1024, inf) != 0)
    if (fgets(Word, 1024, inf) != 0) {
      Word[strlen(Word) - 1] = ' ';
      if (strlen(Word) > 6)
      for (wp = Word; *wp; wp++)
        if (!islower(*wp))
          cont =1 ;
    }
  }
  gp = Known;
  wp = Word;
  while (*wp) {
    *gp = '-';
    gp++;
    wp++;
  }
  *gp = ' ';
}

/*
 * abs:
 *     Return the absolute value of an integer
 */
abs(i)
int   i;
{
  if (i < 0)
  return  -i;
  else
  return i;
}

/*
 * playgame:
 *     play a game
 */
playgame()
{
  register int   *bp;

  getword();
  Errors = 0;
  bp = Guessed;
  while (bp < &Guessed[26]) {
    *bp = 0;
    bp++;
    }
  while (Errors < 7 && index(Known, '-') != 0) {
    prword();
    prdata();
    prman();
    getguess();
  }
  endgame();
}

/*
 * prdata:
 *     Print out the current guesses
 */
prdata()
{
  int    *bp;

  printf("Guessed:  ");
  bp = Guessed;
  while (bp < &Guessed[26])
    if (*bp++)
      putchar((bp - Guessed) + 'a' - 1);
  putchar('0);
  printf("Word #:  %d0, Wordnum);
  printf("Current Average:  %.3f0,
         (Average * (Wordnum - 1) + Errors) / Wordnum);
  printf("Overall Average:  %.3f0, Average);
}

/*
 * prman:
 *     Print out the man appropriately for the give number
```

```
 *     of incorrect guesses.
 */
prman()
{
   int    i;
   char line[9][100];
   char **sp;

   i = 0;
   for (sp = Noose_pict; *sp != 0; sp++) {
      strcpy(line[i], *sp);
      strcat(line[i], "          ");
      i++;
   }

   for (i = 0; i < Errors; i++)
      line[Err_pos[i].y][Err_pos[i].x] =  Err_pos[i].ch;

   for (i = 0; i < 9; i++) {
      printf(line[i]);
      putchar('0');
   }

}

/*
 * prword:
 *     Print out the current state of the word
 */
prword()
{
   printf("Known:  %s0, Known);
}

/*
 * setup:
 *     Set up the strings on the screen.
 */
setup()
{
   register char      **sp;
   int timeval;

   for (sp = Noose_pict; *sp != 0; sp++) {
      printf(*sp);
      putchar('0');
   }

   timeval = time(0);
   srand(timeval + getpid());
   Count = (timeval >= 714332438);
     /* Aug 20, 1992 10:45 AM */
   if ((Dict = fopen("/usr/dict/words", "r")) == 0) {
      perror("/usr/dict/words");
      exit(1);
   }
   fstat(fileno(Dict), &sbuf);
   Dict_size = sbuf.st_size;

}
```