



Reasoning® Inspection Service

Reasoning
Inspection Service
Security Vulnerability Data

Sendmail

v 8.13.0.PreAlpha4

OPEN SOURCE

29-Nov-2004



P.O. Box 478

Menlo Park, CA 94026-0478

+1 650-324-2510

www.reasoning.com

Table of Contents

Introduction	3
Reasoning Inspection Services	3
Types of Inspections	3
Deliverables	3
Inspection Summary	4
Inventory Summary	4
Vulnerability Summary	4
Understanding Security Data Reports	5
Detailed Security Data Reports	6
Understanding Security Vulnerabilities	13
Buffer Overflow	14
Tainted Data	16
Race Condition	18
Risky Operation	20
Weak Random Number Generator	21
Poor Temporary File Name	22
Executing External Program or Dynamic Library	23

Introduction

Reasoning Inspection Services

Reasoning Inspection Services for Java, C, and C++ provide essential expertise that boosts the productivity of development teams by finding software reliability defects and security vulnerabilities faster, earlier, and at a far lower cost than traditional approaches.

The Reasoning service provides many of the benefits of a manual code review, but in significantly less time and at a dramatically lower cost. Reasoning detects and diagnoses reliability defects and security vulnerabilities well before they become costly problems and provides a roadmap to the exact location for remedy and resolution.

Reasoning achieves these ends by automating software inspection, which enhances the organization's existing QA processes and bolsters efforts to provide easy-to-support, high-quality software. As a result, Reasoning enables fast time to ROI and permits software developers to focus on their core competency: software development.

Types of Inspections

Reasoning provides two types of inspection services:

- The *Reliability Inspection Service* identifies defects that may cause an application to crash or behave unpredictably.
- The *Security Inspection Service* identifies security vulnerabilities, through which an attacker may gain unauthorized access.

Deliverables

Reasoning provides two types of reports at the conclusion of the inspection process:

- The *Data Report* makes defect or security vulnerability analysis and repair simple by identifying the type and location of every defect or security vulnerability and describing the circumstances under which they will occur. By providing this map to discovered defects and security vulnerabilities, development time can be spent more effectively.
- The *Metrics Report* is designed for development managers. Providing better insight into problem areas within an application, it enables managers to better plan testing and development efforts.

This is the Security Vulnerability Data report.

Inspection Summary

Inventory Summary

Below is a tabular summary of the application inventory. It lists the number of source and user include files included in the inspection. It also lists the number of source and user include lines of code in the inspection.

Reasoning inspected an incomplete application. Approximately 29 include files were missing. After consultation with the client project manager, it was decided that it was better to proceed with the inspection than to wait for additional include files.

	Files	Lines
Source Code Files	164	79,411
User Include Files	38	4,107
Total	202	83,518

Vulnerability Summary

The column *Vulnerability Instances* in the table below details, per vulnerability category, the number of vulnerabilities.

The column *Vulnerability Density* details, per vulnerability category, the number of vulnerabilities per KLOC, an important measure for comparisons between different applications or development organizations.

The column *Files Affected* details, per vulnerability category, the number of files in the application that have one or more vulnerabilities. Because files may have vulnerabilities in more than one category, the total number of files affected may be less than the sum of the number of files affected per category.

	Vulnerability Instances	Vulnerability Density	Files Affected
Buffer Overflow	1	0.01	1
Race Condition	0	0.00	0
Risky Operation	0	0.00	0
Tainted Data	6	0.08	4
Total	7	0.09	5

Understanding Security Data Reports

The Security Data Report provides a detailed description of each security vulnerability found by Reasoning. The purpose of this description is to simplify defect analysis and repair by supplying the key information relevant to each vulnerability.

The report provides six items for each vulnerability detected:

- **Vulnerability Category:** Type of security vulnerability detected
- **Vulnerability ID:** Unique number to identify the vulnerability
- **Location:** File name and line number where the vulnerability is located
- **Description:** Written explanation of the vulnerability
- **Conditions:** Set of conditions that will cause the vulnerability to occur
- **Code Fragment:** Fragment of the code that contains the vulnerability

More information on the vulnerability categories is found in the "Understanding Security Vulnerabilities" section at the end of this report.

Vulnerability Category Buffer Overflow**Vulnerability ID** 00-0001**Location** /contrib/oldbind.compat.c: 53**Description** Check to be sure that argument 2 passed to the function **strcpy** () on line **53** will not copy more data than can be handled, allowing the buffer **dbuf** declared on line **51**, to overflow. A similar error can be found on line 56.**Conditions** None.**Code Fragment**

```
41 res_querydomain(host, dname, class, type, data, datalen)
42     char *      host;
43     char *      dname;
44     int         class;
45     int         type;
46     char *      data;
47     int         datalen;
48 {
49     int         n;
50     querybuf buf;
51     char       dbuf[256];
52
53     strcpy(dbuf, host);
54     if (dbuf[strlen(dbuf)-1] != '.')
55         strcat(dbuf, ".");
56     strcat(dbuf, dname);
```

Vulnerability Category Tainted Data**Vulnerability ID** 00-0002**Location** /mail.local/mail.local.c: 355**Description** The function **getlogin()** called on line **355** returns information that can be controlled by a local user, so don't trust it for security purposes.**Conditions** None.**Code Fragment**

```
172  int
173  main(argc, argv)
174      int argc;
175      char *argv[];
176  {
177      struct passwd *pw;
178      int ch, fd;
179      uid_t uid;
180      char *from;
181      char *mbdbname = "pw";
182      int err;
183      extern char *optarg;
184      extern int optind;
...
354      uid = getuid();
355      if (from == NULL && ((from = getlogin()) == NULL ||
356          (pw = getpwnam(from)) == NULL ||
357          pw->pw_uid != uid))
358          from = (pw = getpwuid(uid)) != NULL ? pw->pw_name : "???";
```

Vulnerability Category Tainted Data**Vulnerability ID** 00-0003**Location** /sendmail/conf.c: 1172**Description** The function `getlogin()` called on line **1172** returns information that can be controlled by a local user, so don't trust it for security purposes. This data is used on line **1183**.**Conditions** The conditional expression `myname == NULL` on line **1170** evaluates to **true**.**Code Fragment**

```
1146 /*
1147 **  USERNAME -- return the user id of the logged in user.
1148 **
1149 ** Parameters:
1150 **     none.
1151 **
1152 ** Returns:
1153 **     The login name of the logged in user.
1154 **
1155 ** Side Effects:
1156 **     none.
1157 **
1158 ** Notes:
1159 **     The return value is statically allocated.
1160 */
1161
1162 char *
1163 username()
1164 {
1165     static char *myname = NULL;
1166     extern char *getlogin();
1167     register struct passwd *pw;
1168
1169     /* cache the result */
1170     if (myname == NULL)
1171     {
1172         myname = getlogin();
1173         if (myname == NULL || myname[0] == '\0')
1174         {
1175             pw = sm_getpwuid(RealUid);
1176             if (pw != NULL)
1177                 myname = pw->pw_name;
1178         }
1179         else
1180         {
1181             uid_t uid = RealUid;
1182
1183             if ((pw = sm_getpwnam(myname)) == NULL ||
1184                 (uid != 0 && uid != pw->pw_uid))
```

Vulnerability Category Tainted Data**Vulnerability ID** 00-0004**Location** /sendmail/conf.c: 1236**Description** The function `ttyname()` called on line **1236** returns information that can be controlled by a local user, so don't trust it for security purposes. Similar errors can be found on lines 1236 and 1237.**Conditions** None.**Code Fragment**

```
1203 /*
1204 **  TTYPATH -- Get the path of the user's tty
1205 **
1206 ** Returns the pathname of the user's tty. Returns NULL if
1207 ** the user is not logged in or if s/he has write permission
1208 ** denied.
1209 **
1210 ** Parameters:
1211 **   none
1212 **
1213 ** Returns:
1214 **   pathname of the user's tty.
1215 **   NULL if not logged in or write permission denied.
1216 **
1217 ** Side Effects:
1218 **   none.
1219 **
1220 ** WARNING:
1221 **   Return value is in a local buffer.
1222 **
1223 ** Called By:
1224 **   savemail
1225 */
1226
1227 char *
1228 ttypath()
1229 {
1230     struct stat stbuf;
1231     register char *pathn;
1232     extern char *ttyname();
1233     extern char *getlogin();
1234
1235     /* compute the pathname of the controlling tty */
1236     if ((pathn = ttyname(2)) == NULL && (pathn = ttyname(1)) == NULL &&
1237         (pathn = ttyname(0)) == NULL)
1238     {
1239         errno = 0;
1240         return NULL;
1241     }
```

Vulnerability Category Tainted Data**Vulnerability ID** 00-0005**Location** /sendmail/domain.c: 1204**Description** The value of the environment variable "**HOSTALIASES**", returned by the function **getenv()** called on line **1204**, may include malicious content. This is used to open a file on line **1206** without any checking.**Conditions** None.**Code Fragment**

```
1189 static char *
1190 gethostalias(host)
1191     char *host;
1192 {
1193     char *fname;
1194     SM_FILE_T *fp;
1195     register char *p = NULL;
1196     long sff = SFF_REGONLY;
1197     char buf[MAXLINE];
1198     static char hbuf[MAXDNAME];
1199
1200     if (ResNoAliases)
1201         return NULL;
1202     if (DontLockReadFiles)
1203         sff |= SFF_NOLOCK;
1204     fname = getenv("HOSTALIASES");
1205     if (fname == NULL ||
1206         (fp = safefopen(fname, O_RDONLY, 0, sff)) == NULL)
```

Vulnerability Category Tainted Data**Vulnerability ID** 00-0006**Location** /sendmail/envelope.c: 764**Description** The function `ttyname()` called on line **764** returns information that can be controlled by a local user, so don't trust it for security purposes. This data is stored on line **770**.**Conditions** The conditional expression `TTYNAME` on line **760** evaluates to **true** AND The method `macvalue('y', e)`, called on line **762**, returns **null**.**Code Fragment**

```

693  /*
694  **  INITSYS -- initialize instantiation of system
695  **
696  **  In Daemon mode, this is done in the child.
697  **
698  **  Parameters:
699  **    e -- the envelope to use.
700  **
701  **  Returns:
702  **    none.
703  **
704  **  Side Effects:
705  **    Initializes the system macros, some global variables,
706  **    etc. In particular, the current time in various
707  **    forms is set.
708  */
709
710 void
711 initsys(e)
712     register ENVELOPE *e;
713 {
714     char buf[10];
715 #ifdef TTYNAME
716     static char ybuf[60];          /* holds tty id */
717     register char *p;
718     extern char *ttyname();
719 #endif /* TTYNAME */
720 ...
760 #ifdef TTYNAME
761     /* tty name */
762     if (macvalue('y', e) == NULL)
763     {
764         p = ttyname(2);
765         if (p != NULL)
766         {
767             if (strrchr(p, '/') != NULL)
768                 p = strrchr(p, '/') + 1;
769             (void) sm_strncpy(ybuf, sizeof ybuf, p);
770             macdefine(&e->e_macro, A_PERM, 'y', ybuf);
771         }
772     }

```

Vulnerability Category Tainted Data**Vulnerability ID** 00-0007**Location** /sendmail/envelope.c: 1112**Description** The value of the environment variable "**HOME**", returned by the function `getenv()` called on line **1112**, may include malicious content.**Conditions** The conditional expression `bitnset(M_HASPWENT, e->e_from.q_mailer->m_flags)` on line **1050** evaluates to **false** AND The conditional expression `internal && OpMode != MD_DAEMON && OpMode != MD_SMTP` on line **1108** evaluates to **false** AND The conditional expression `e->e_from.q_home == NULL` on line **1110** evaluates to **true**.**Code Fragment**

```

931 void
932 setsender(from, e, delimptr, delimchar, internal)
933     char *from;
934     register ENVELOPE *e;
935     char **delimptr;
936     int delimchar;
937     bool internal;
938 {
939     register char **pvp;
940     char *realname = NULL;
941     char *bp;
942     char buf[MAXNAME + 2];
943     char pvpbuf[PSBUFSIZE];
944     extern char *FullName;
945     ...
1050     if (bitnset(M_HASPWENT, e->e_from.q_mailer->m_flags))
1051     {
946     ...
1107     }
1108     else if (!internal && OpMode != MD_DAEMON && OpMode != MD_SMTP)
1109     {
1110         if (e->e_from.q_home == NULL)
1111         {
1112             e->e_from.q_home = getenv("HOME");
1113             if (e->e_from.q_home != NULL)
1114             {
1115                 if (*e->e_from.q_home == '\0')
1116                     e->e_from.q_home = NULL;
1117                 else if (strcmp(e->e_from.q_home, "/") == 0)
1118                     e->e_from.q_home++;
1119             }
1120         }

```

Understanding Security Vulnerabilities

This section provides a detailed explanation of each of the security vulnerability categories that Reasoning finds in C/C++ applications. These explanations support evaluation of the impact of each of the vulnerabilities listed in the Security Data Report. For each vulnerability category, this section provides:

- A short description of the category
- The possible impact of an exploitation of the vulnerability category
- An example code fragment taken from open source projects that shows an instance of the vulnerability
- Advice on how the vulnerability may be repaired

A *false positive* is code that on the surface looks like a security vulnerability, but can be proven to be unexploitable. Reasoning makes every attempt to avoid reporting technical false positives, which can be proven to be secure without resorting to domain knowledge. This may leave reported vulnerabilities that can be proven to be false positives based on domain knowledge. To make it easier to assess the reported vulnerabilities, the data report provides as much context as possible.

Do not discard vulnerabilities because it seems unlikely that the vulnerable code path will ever be executed with data that would expose the vulnerability. The approach of relying on the low likelihood of someone discovering the vulnerability is called "security by obscurity" and doesn't work. It only takes one smart person to find a vulnerability and design an exploit for it. Once the exploit has been designed, it may be published in the back alleys of the Internet and used by many attackers, even if their skill level is much lower than that of the designer of the exploit.

Over 70% of the effort spent on most software applications is spent in maintenance. Even though the vulnerability may be a false positive based on domain knowledge, it may still become an actual vulnerability through future changes to the surrounding code. Removing these vulnerabilities will reduce the future cost and risk associated with maintaining the application:

- training new staff and consultants
- extending the system with new functionality, for example, web-enabling it
- rewriting or replacing the system

This document does not attempt to replace the extensive literature describing security vulnerabilities, their impact, and techniques to remove or prevent them. The references below can serve as a starting point for further education:

- *How to Break Software Security*, by Herbert H. Thompson and James A. Whittaker, 2003.
- *Writing Secure Code*, by Michael Howard and David LeBlanc, 2002.
- *Security Engineering: A Guide To Building Dependable Distributed Systems*, by Ross J. Anderson, 2001.
- *Practical UNIX & Internet Security, 3rd Edition*, by Gene Spafford, Simson Garfinkel, and Alan Schwartz, 2003.
- *Secure Programming for Linux and UNIX HOWTO*, by David A Wheeler, <http://www.dwheeler.com/secure-programs>, 2003.

Buffer Overflow

Description

A *buffer overflow* occurs when a program or process tries to store more data in a *buffer* (temporary data storage area) than it can hold. The extra information—which has to go somewhere—can overflow into adjacent buffers, overwriting and corrupting the valid data held in them. Although buffer overflows may occur accidentally through programming error, they are now the most common threat to software security. In buffer overflow attacks, the extra data may contain code designed to, for example, damage the user's files, change data, or disclose confidential information. The root cause of buffer overflow attacks is the lack of bounds checking in the C and C++ programming languages (and C# executed with the “unsafe” switch), combined with poor programming practices.

In July 2000, a buffer overflow vulnerability was discovered in Microsoft Outlook and Outlook Express. It could be exploited by simply sending an e-mail message to a vulnerable computer. A defect in the program's message header processing made it possible for attackers to execute arbitrary code on the recipient's computer. Because this happened as soon as the recipient downloaded the message from the server, victims could not defend themselves against the virus by simply discarding the message. Microsoft has created a patch to remove the vulnerability.

Impact

The impact of a buffer overflow can be any of the following:

- An attacker may exploit the buffer overflow to take control of the software. The impact depends on the nature of the executable code.
- If the vulnerable software runs with administrator permissions, the malicious code may even take over the entire system.
- An attacker may exploit the buffer overflow to crash the application, for example, in the context of a Denial of Service (DoS) attack. This requires little sophistication; just providing random data larger than the software can handle will cause a crash.

Example

```
49 int finddomain(char *host, char *dname) {
50
51     char dbuf[256];
52     ...
57     if (dbuf[strlen(dbuf)-1] != '.')
58         strcat(dbuf, ".");
59     strcat(dbuf, dname);
```

In the code fragment above, the call to the function `strcat()` is supposed to concatenate the contents of the string `dname` to the string that is already in the array `dbuf`, namely the concatenation of `host` and a period. If the function `finddomain()` is called with excessively long hostnames, domain names, or both, such that the sum of their lengths is larger than 254, the buffer `dbuf` will overflow, overwriting the call stack with the excessive data.

Repair

Depending on the context, this vulnerability can be repaired in one of several ways:

- Allocate buffers dynamically after the size of the data they need to contain is known. For example:

```
char *newBuffer;
newBuffer = malloc(strlen(oneFile)+strlen(twoFile)+30);
if (!newBuffer) return;
sprintf(newBuffer, ``can not copy file %s to %s``, oneFile, twoFile);
```

The `sprintf()` cannot cause `newBuffer` to overflow because `newBuffer` is allocated sufficiently large to contain both file names and the constant text of the error message.

- Use the safe string manipulation functions, such as `strncat()` and `snprintf()`, of the standard C Library. These functions have an extra argument that specifies the length of the buffer. They are also known as “n” versions because their name has an additional “n” character. For example:

```
char newBuffer[BUFSIZE];
snprintf(newBuffer, sizeof newBuffer, ``can not copy file %s to %s``,
         oneFile, twoFile);
```

- Check for string lengths before writing to the buffer:

```
char format = ``can not copy file %s to %s``;
if (strlen(oneFile) + strlen(twoFile) + strlen(format)
    >= sizeof newBuffer)
    return ``error: buffer not big enough``;
sprintf(newBuffer, format, oneFile, twoFile);
```

- Use the “%Ns” format string instead of “%s”:

```
char buffer[1000];
sprintf(buffer, ``Bad file is %985s``, filename);
```

Tainted Data

Description

Whenever a software program obtains data from the outside world, it needs to validate that the data is within the design specifications of the program. As long as data is not validated, it is called *tainted data*. The use of tainted data may cause the program to perform operations that do not conform to the design of the program. The functions `getenv()`, `gethostbyname()`, `catgets()`, and `dgettext()` are examples of functions that return data that can be controlled by an attacker, and therefore is tainted data.

Impact

Tainted data can cause buffer overflows directly and indirectly. A direct buffer overflow is caused by tainted data that is too large to fit in buffers supposed to contain it. An indirect buffer overflow occurs when the tainted data is used as a format string to functions such as `printf()`.

A subtler exploit of tainted data does not overflow buffers, but misleads the program or its user. This is sometimes called “spoofing”.

For programs that are run with special (administrator or “root”) permissions, there are additional impacts. When provided with tainted data, these programs may enable users to read or write files they are not allowed to read or write. The same danger exists for changing the permissions, history, or ownership of files and processes.

The business impact depends on the usage context of the data and the creativity of the attacker. It includes stolen credit card numbers and expiry dates, identity theft, unauthorized money, defacement of the company's public image, and corruption of essential business data such as order information and payment records. The attacker may be external or internal to the business running the software program.

Example

```
1190 FILE *openhost(char *host) {
    ...
1193     char *hname;
1194     SM_FILE_T *fp;
    ...
1204     hname = getenv("HOSTNAME");
1205     if (hname == NULL ||
1206         (fp = openhfile(hname, READ|WRITE)) == NULL)
1207         return NULL;
```

The function call `getenv()` on line 1204 obtains the value of the environment variable `HOSTNAME`, which is intended to contain the name of the computer on which the application runs, which is then used on line 1206 to open a file with the name of the computer. However, the value of `HOSTNAME` could be changed to contain the full pathname of a file that is crucial for the operation of the system. The program would then start reading and writing from that file, creating havoc.

Repair

The most robust way to repair tainted data vulnerabilities is to read information from trusted sources only. The next best technique is to validate the data, that is, to verify that it is not too long and does not contain malicious or false content.

For `getenv()`, you need to double check that you really want the user to control whatever it is that the environment variable is controlling. This is especially true of programs that run with special permissions higher than the permissions of the users running them.

For the string catalog functions, such as `catgets()` and `dgettext()`, you should verify that the environment variables they use have reasonable data in them and the data files they read have not been tampered with. The latter may be achieved by checking the data files against checksums or signatures.

Defend against formatting characters in tainted data by providing your own format string:

```
10 printf(`Hello `);
11 /* printf(name); */
12 printf(`%s`, name);
```

If an attacker would enter the formatting string “%100000s” as his name, the commented statement on line 11 would print 100,000 characters from memory, possibly containing confidential information or hints of additional vulnerabilities. The statement on line 12 will just print “%100000s”.

Race Condition

Description

Time elapses between the verification by a software program that a planned operation (such as reading or writing a file) is safe, and the execution of the operation itself. In this time period, attackers may change something in the program's environment—for example, the contents or access restrictions of a file—that makes the execution of the operation unsafe.

These attacks are sometimes called TOCTOU (“tock-toe”) because they involve “Time Of Check” versus “Time Of Use” of the file system. In order to exploit a race condition, an attacker needs to have a relatively high level of control over the environment. For this reason, attacks based on race conditions are typically performed by insiders or by attackers who have already gained access through an unrelated security hole but want more permissions than they have.

Impact

The impact depends on the character of the operation that is executed, but usually includes the ability to write to files that the attacker could not modify otherwise. This can lead to changing passwords (for example).

Example 1

```
2903     /* Is file there? */
2904     if (stat(pfilename, &stat_buf) == -1)
2905     {
    ...
2908     }
2909
2910 #ifndef WIN32
2911     /* Permissions OK? */
2912     if (stat_buf.st_mode & (S_IRWXG | S_IRWXO))
2913     {
    ...
2918         return NULL;
2919     }
2920 #endif
2921
2922     fp = fopen(pfilename, "r");
```

The code above calls `stat()` to check existence of a password file whose name is stored in the variable `pfilename`. Later, the program calls `fopen()` on a file with the same name, but at that point it might be a different file.

Example 2

```
133 /* Read filename into an allocated buffer and return it. */
134 static char *eat_file (filename)
135 char *filename;
136 {
137     struct stat finfo;
138     size_t file_size;
139     char *buffer;
140     int i, file;
141
142     if ((stat (filename, &finfo) < 0)
143         || (file = open (filename, O_RDONLY, 0666)) < 0)
144         return ((char *)NULL);
145
146     file_size = (size_t)finfo.st_size;
147     ...
148     /* Read the file into BUFFER. */
149     buffer = (char *)safemalloc (file_size + 1);
150     i = read (file, buffer, file_size);
151     close (file);
```

The function above attempts to read an entire file into a buffer. However, because the size of the file is gotten via a `stat()` call, while the actual file is opened via a subsequent `open()` call, it is possible for an attacker to replace the first file by a larger file, causing a buffer overflow.

Repair

Avoid the use of the functions that may contribute to race conditions. In many cases, there are safer alternatives. For example, `fstat()` is a safer alternative to the `stat()` above, but it must be used after the `fopen()`.

Race conditions involving the function `access()` are particularly hard to fix, because there is no `faccess()` function (nor can there be: `access()`'s actions are path dependent, not file descriptor dependent). This function is typically used in `setuid` programs, which are common targets of race condition attacks. Using the `seteuid()` and `setegid()` functions allows you to use the `open()` call directly, so the `access()` function is not needed.

Risky Operation

Reasoning's service finds three types of risky operations that, depending on the context, may cause security vulnerabilities:

- Weak random number generator
- Poor temporary file name
- Executing external program or dynamic library

In the sections below, each of these vulnerabilities is described. Because examples of code containing these vulnerabilities would be of much less generic value than the examples provided for buffer overflows, tainted data, and race conditions, they are omitted from the descriptions.

Weak Random Number Generator

Description

Using a *weak random number generator* means that your “random” actions (or data) are not as random as they should be; an attacker may be able to predict what your program will do next or what data it will use.

Impact

In the worst case, an attacker may guess the key used to encrypt sensitive data and use that key to decipher the data or trick the software into thinking an untrustworthy process should be trusted.

If the weak random number generator is used to create a temporary file name, then you have inherited all the problems of a poor temporary file name (see next page).

Repair

Use a better random number generator. There is no safe random number generator shipped with all versions of Windows or UNIX, but there are many available as open source or from third party developers.

The best “standard” random number generator is `random()`. Although it is not great, it is better than `rand()` or any of the `*rand48()` functions, so use it in preference to any of those. If you must use `random()`, then don't use `srandom()` to initialize it, but rather use `initstate()` and take advantage of the extra initialization data it provides.

Poor Temporary File Name

Description

Using a poor temporary file name, in a way that allows an attacker to predict the name ahead of time, is dangerous because it allows an attacker to use the program's private data.

Impact

This can allow an attacker to read data from a temp file, or write data to it, that will then be used by the program. Especially on UNIX file systems, where named pipes and file system sockets can live in the file system and look just like “plain” files, an attacker has a lot of tools to steal data or spoof your program with phony data.

Repair

Depending on the sensitivity of the data in the temporary file, you can use one or more of the techniques described below:

- Use the best temporary file name generator available, use flags to create the most random file names, and use the best random number generator available.
- Do not use an already existing temp file, and if you must use it, at least truncate it and make sure it is a plain file before using it.
- Encrypt the data in the temporary file, or calculate and confirm a checksum for the file.

Executing External Program or Dynamic Library

Description

Executing an external program or dynamic library is risky because it is possible that an attacker has replaced the program or library.

Impact

For a program running with high permission levels (as root, system, admin, etc.) the effects can be devastating. The attacker can obtain total control over the machine, change data or programs, and generally compromise the computer and all data on it.

Even for a program running with standard user permissions, however, there can be bad effects because the user running the program may have access to sensitive data files. An attacker can use this vulnerability to read private files.

Repair

There are several techniques to increase the safety of executing an external program or dynamic library:

- Use the best function to execute a program or to load a dynamic library.
- Always specify full path names, not relative ones.
- When possible, check an executable file and the file system before using it.

Using the best function usually means using the one that does the *least*. Many programmers naturally choose the most powerful function they can find, but this leads to lower security. For example, don't use a function that searches a file system for an executable or library. An attacker can put a malicious program or library earlier in the search path than the intended one. For the same reason, don't use a function that will automatically append an extension to the file name. Executing a program directly is safer than starting a shell to execute it, because the shell introduces a whole new layer of potential vulnerabilities. Whenever possible, use full path names to prevent searching, even if the function you are using will search for your program or library.

Finally, verify that the external program or dynamic library you are loading is the exact one you expect to load, if this is possible. Use a checksum, or make sure the contents are what you expect. If this is not possible, you may be able to verify the general type of binary. Finally, you can check the file permissions, both of the binary itself, and the directories that contain it, to see if they allow replacement, and if the ownership is what you expected.



P.O. Box 478

Menlo Park, CA 94026-0478

+1 650-324-2510

www.reasoning.com