

# SMARTRISK™ ANALYZER USER'S GUIDE

**May 2004**

**Release 2.0**

## **BOSTON**

@stake, Inc.  
196 Broadway  
Cambridge, MA 02139  
United States of America

Phone: +1 617.621.3500  
Fax: +1 617.621.1738  
<http://www.atstake.com/>

## **LONDON**

Atstake Limited  
6 - 8 James Street  
London W1U 1ED  
United Kingdom

Phone: +44(0)20.7495.7002  
Fax: +44(0)20.7495.7062  
<http://www.atstake.co.uk/>

**CHICAGO   NEW YORK   RALEIGH   SAN FRANCISCO   SEATTLE**

© 2001-2004 @stake, Inc. All rights reserved.

@stake and SmartRisk are registered trademarks of @stake, Inc. All other product or company names are trademarks or registered trademarks of their respective companies.

### **Limited Warranty; Disclaimer.**

Licenser will replace, at no charge, defective media that are returned within ninety (90) days of shipment. Licenser warrants, for a period of ninety (90) days from the shipment date, that the Software will perform in substantial compliance with the written materials accompanying Software on that hardware and operating system software for which it was designed, as stated in the documentation. If, within such ninety (90) days period, Licensee reports to Licenser that Software is not performing as described above, Licenser will, at its option, repair or replace the Software. The foregoing states the entire liability of Licenser with respect a breach of the warranty set forth herein. EXCEPT AS OTHERWISE PROVIDED HEREIN, THE SOFTWARE AND THE DOCUMENTATION ARE BEING SUPPLIED TO LICENSEE ON AN "AS IS" BASIS. LICENSOR HEREBY EXPRESSLY DISCLAIMS ALL WARRANTIES REGARDING THE SOFTWARE AND THE DOCUMENTATION, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, AND TITLE, AND ALL WARRANTIES IMPLIED FROM ANY COURSE OF DEALING OR USAGE OF TRADE. EXCEPT AS SPECIFICALLY SET FORTH HEREIN, LICENSOR DOES NOT WARRANT THAT (A) THE SOFTWARE WILL MEET LICENSEE'S REQUIREMENTS, (B) OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR FREE, OR (C) DEFECTS WILL BE CORRECTED



# TABLE OF CONTENTS

<b>PREFACE:</b>	<b>Quick Start .....</b>	<b>ix</b>
<b>CHAPTER 1:</b>	<b>Getting Started</b>	
	About SmartRisk Analyzer .....	1-1
	System Recommendations .....	1-2
	Contact @stake .....	1-2
	Support Information .....	1-3
<b>CHAPTER 2:</b>	<b>SmartRisk Analyzer Overview</b>	
	Operating SmartRisk Analyzer .....	2-3
	Supported Platforms and Environments .....	2-5
	Solaris Platforms .....	2-5
	Windows Platforms .....	2-5
	Deep Binary Modeling .....	2-7
	Analyzing the Application .....	2-9
	Reporting .....	2-13
	Workflow .....	2-16
	User Interface Menu Options .....	2-17
<b>CHAPTER 3:</b>	<b>Installation</b>	
	Installing SmartRisk Analyzer .....	3-1
	Uninstalling SmartRisk Analyzer .....	3-2
<b>CHAPTER 4:</b>	<b>Sample Project Walk-Through</b>	
	Walk-Through Using SmartRisk Analyzer .....	4-1
	Determining Severity Differences .....	4-11
	Generating Reviewer Files .....	4-12
	Making Notes in SmartRisk Analyzer .....	4-13
	Viewing a Created Sticky Note .....	4-14
	Adding Creators .....	4-14
<b>CHAPTER 5:</b>	<b>Performing Security Analyses</b>	
	Analyzing an Application .....	5-1
	Creating a SmartRisk Project File .....	5-1
	Binary Modeling .....	5-3
	Security Scans .....	5-3
	Reviewing Results .....	5-7

	Editing Results.....	5-7
	Exporting Results.....	5-7
	Interpreting SmartRisk Results.....	5-7
	Prioritizing Security Vulnerabilities.....	5-8
	Interactive Exploration .....	5-8
	Saving an Analysis.....	5-9
	SmartRisk Environment Files .....	5-10
	Selecting Existing Environment Files.....	5-10
<b>CHAPTER 6:</b>	<b>Reporting</b>	
	Getting a Report.....	6-1
	Summary Reports .....	6-1
	Detailed Reports.....	6-3
	Exporting Results to a Generated Report .....	6-3
	Reading Reports .....	6-4
<b>CHAPTER 7:</b>	<b>Understanding Error Messages</b>	
	Types of Severity.....	7-1
	Determining Severity .....	7-2
	Prioritizing Severity .....	7-2
	Severity Descriptions .....	7-2
	Multiple Errors in Code .....	7-5
<b>CHAPTER 8:</b>	<b>SmartRisk Analyzer Reference</b>	
	Scan Types .....	8-1
	Stack/Heap Buffer Overruns .....	8-3
	Format String Vulnerabilities .....	8-4
	Error Return Checking.....	8-5
	Integer Overflows/Underflows .....	8-6
	Threading/Race Conditions.....	8-8
	Cryptography.....	8-9
	Database .....	8-10
	Denial of Service .....	8-11
	Reliability Issues.....	8-13
	Input Validation .....	8-14
	Privilege Escalation.....	8-15
	Network Issues.....	8-16
	Common Triggers as Security Risks .....	8-17
	gets .....	8-18
	Memory Functions.....	8-19
	memcpy.....	8-21
	printf .....	8-22
	rand .....	8-24
	recv .....	8-25
	scanf.....	8-26
	strcpy.....	8-27
	syslog .....	8-28
	umask.....	8-29
<b>APPENDIX A:</b>	<b>SmartRisk Troubleshooting</b>	
<b>APPENDIX B:</b>	<b>Recommended Reading</b>	

APPENDIX C: Glossary

INDEX



# PREFACE: QUICK START

SmartRisk™ Analyzer is a breakthrough solution for identifying security flaws in software applications. Below are quick instructions to run SmartRisk Analyzer. For more in depth instructions and details on SmartRisk Analyzer, please refer to other sections in this user's guide.

1. Determine the executable file you want to analyze for security risks. For more information on preparing your executable, please refer to [page 2-5](#).
2. From the **Start** menu, run **SmartRisk Analyzer** by clicking the SmartRisk icon in the appropriate **Programs** folder.
3. In the first dialog, select **Create a New Project**. Click **OK**.
4. Select **New Binary Analysis**, then browse to the folder where your executable and source code is stored, and name your SmartRisk Project file. Click **OK**.

**Note:** You must store SmartRisk Project files in the upper level directory of your analyzed executable.

5. For the **New Binary Analysis**, browse to find your executable. Click to open the executable, and SmartRisk inputs the **Platform** and **Compiler** information. Click **Next**.
6. **Select Environments** you wish to analyze, then click **Finish**.

**Note:** SmartRisk selects environments to analyze by reading the executable for compatibility information. Deselecting these may result in a poor analysis.

7. Once your executable is loaded, build the binary model by selecting **Start Binary Analysis** in the **Project** menu. Binary Analysis can take several hours for large executable files.
8. When the analysis finishes, **Save** the **Project** from the **File** menu. You do not need to rebuild the model, unless the executable changes.
9. Scan the Project for the risk analysis by running a **FullSecurityScan** from the **Scans** menu under **Global Scans**. Once the scan finishes, the **Summary Report** and **Detailed Report** reveal the discovered security flaws.
10. Save the report as a read-only file under the **Reader** menu and select **Generate Reader Document** for distribution. You may also begin investigating the discovered errors by clicking through the **Annotation List**.

For additional information on SmartRisk Analyzer, refer to the following:

- For System Recommendations, Contact Information, and Support Information, refer to [Chapter 1](#).
- For installation instructions, see [Chapter 3](#).
- For a detailed walk-through of SmartRisk Analyzer using a sample application, refer to [Chapter 4](#).
- For information on Reports, refer to [Chapter 6](#).
- For Troubleshooting, refer to [Appendix A](#).

# CHAPTER 1: GETTING STARTED

SmartRisk™ Analyzer is a software security tool that enables you to identify security flaws in software applications before the code is released to the customer. Built from the expertise of software security consultants, SmartRisk Analyzer locates security flaws in applications, makes suggestions to improve the code with security flaws, and maintains security awareness for developers in their code while building applications.

## About SmartRisk Analyzer

SmartRisk Analyzer is a Windows®-based program that analyzes the code security of applications developed for Windows® and Solaris™ platforms. Using a Visual Studio®-style interface, SmartRisk Analyzer finds implementation security flaws, evaluates relative security flaws based on context and use, prioritizes the results, and makes suggestions for remediation.

SmartRisk Analyzer helps locate security flaws in your application before software release or deployment, cutting down on the number of future patch releases and security fixes.

There are two components of SmartRisk Analyzer. The first is **Analyzer**. Analyzer runs the full security scans and reports flaws and vulnerabilities, after which you can save the results to a **SmartRisk Analyzer Database Document** to be reviewed in the second component, **Reviewer**. Reviewer is used to review the read-only files, share and make notes, and use as a reference. Reviewer does not run security scans or report flaws.

SmartRisk Analyzer does not perform source code style checking, in that it does not point out performance flaws, typos, and unusual code styling in the application. Likewise, SRA does not verify that the implementation of an application matches the design, nor does it perform black-box penetration testing.

## System Recommendations

@stake recommends the following system for SmartRisk Analyzer installation and operation:

- Windows XP Professional
- 3GHz CPU
- 3GB RAM
- 10 GB disk space

**Note:** SmartRisk Analyzer can only be installed on Windows. Please see [Chapter 3](#) for installation instructions.

### Supported Environments

SmartRisk Analyzer locates security problems in applications compiled on Windows and Solaris platforms using the following programming languages:

- C
- C++
- Java™ (J2EE)

### Package Contents

You should have received the following items from @stake with your SmartRisk Analyzer:

- SmartRisk Analyzer Installation CD

If you are missing any of these pieces, please contact @stake immediately.

## Contact @stake

@stake is headquartered in Cambridge, Massachusetts at the following address:

### In North America

@stake, Inc.  
196 Broadway  
Cambridge, MA 02139  
United States of America

T +1 617 621 3500  
F +1 617 621 1738  
<http://www.atstake.com>

### In Europe

Atstake Limited  
6-8 James Street  
London W1U 1ED  
United Kingdom

T +44 (0)20 7495 7002  
F +44 (0)20 7495 7062  
<http://www.atstake.co.uk>

@stake also has regional offices in Chicago, New York, Raleigh, San Francisco, and Seattle. You can contact @stake toll free at 1.866.621.3500.

## Support Information

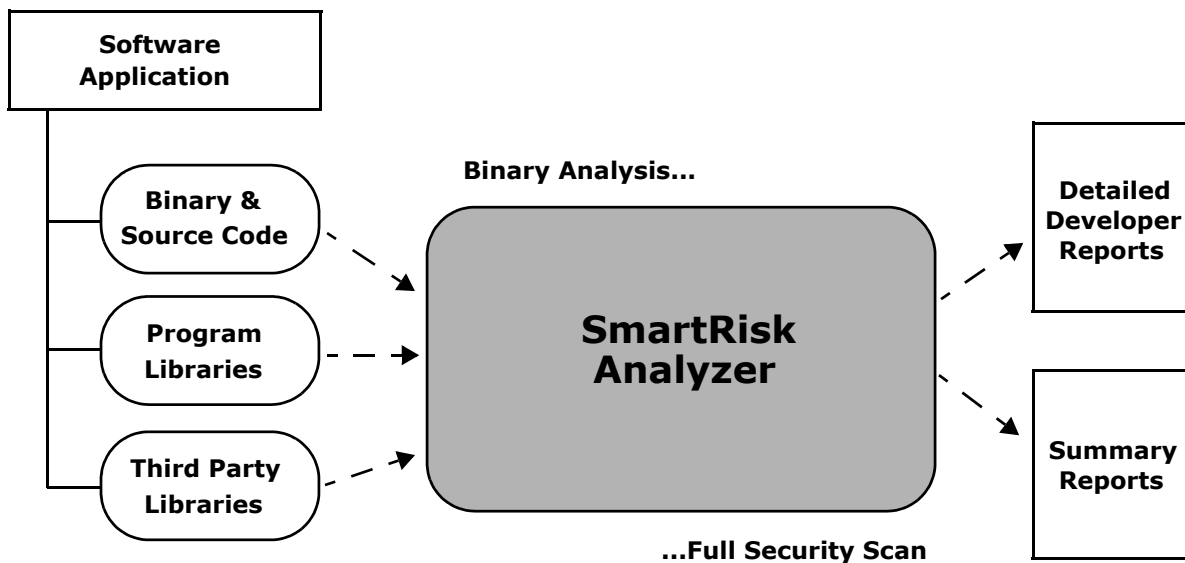
@stake makes every effort to deliver a high quality product, however, in the event you need technical support, you can contact @stake during regular business hours, Monday – Friday, 9 AM to 5 PM EST at 1.866.621.3500 or at [support@atstake.com](mailto:support@atstake.com).



# CHAPTER 2: SMARTRISK ANALYZER OVERVIEW

SmartRisk Analyzer (SRA) locates security flaws in software applications by analyzing the binary files in your source code. This chapter gives an overview of SmartRisk Analyzer, and explains how SRA detects security flaws in your applications.

The diagram below illustrates the operation of a typical project.



**Figure 1: The Operation of SmartRisk Analyzer**

SmartRisk Analyzer creates projects using your developed software applications. SRA conducts a **Binary Analysis** and **Full Security Scan** of the application and the supporting files. At completion, SRA produces detailed and summary reports of security vulnerabilities found, describing each by severity and offering suggestions for fixes.

SmartRisk Analyzer can locate security flaws based upon the platform and

environment the application runs in. The platform and environment are discovered at the creation of an SRA project. Code is analyzed and checked against a database of potential flaws that could result in security violations once the application is deployed in its runtime environment. The results are reported in five levels of severity with suggested fixes, intended to point out security flaws, and help developers find efficient solutions for fixes.

Once generated, reports may be saved to a **SmartRisk Analyzer Database Document** read-only file, or **\*.srf file**. Reports are viewable by both the **Analyzer** and **Reviewer** versions of SmartRisk Analyzer. The Reviewer version of SmartRisk Analyzer can display a summary of security leaks found through analyzer, but the Reviewer does not operate the **Binary Analysis** and **Full Security Scan** that finds these vulnerabilities. The Reviewer tool enables developer groups to run Analyzer on fewer systems and away from developer environments. Developers then share the reviewer files to view the vulnerabilities.

The Reviewer summary then displays and ranks the degrees of severity in the found security vulnerabilities. A read-only version of the code with an output window and sticky notes are also available.

# Operating SmartRisk Analyzer

The SmartRisk Analyzer interface reflects a Visual Studio-style design. Operating SmartRisk Analyzer follows five steps to locating security flaws in your application:

- Load full source code and compiled binary
- Run the Binary Analysis
- Run Security Scans
- Export to Review Data (the SRA Reviewer)
- Analyzing Output Data

## Loading full source and compiled binary

Create a new project when SmartRisk Analyzer asks for a name for your new \*.saf project, then a dialog window asks to locate your application's executable. All supporting files for that application should reside in the same central location, because SmartRisk uses the links inside the application to build a model. Once the application is loaded, run a **Binary Analysis**.

Your application must be a **complete compiled debug build** of the software. Loading an application that is not compiled with full debugging information into SmartRisk Analyzer results in an error, and SmartRisk Analyzer can not properly analyze the application.

Likewise, all library files and other supporting pieces of code your application calls on must be available. If a piece of code, such as a library file is missing, SmartRisk Analyzer indicates the file is missing and requests that you copy the file into the same directory as your application.

**Note:** Load only complete compiled debug build of the software. Failure to do so results in an error.

## Run the Binary Analysis

**Start Binary Analysis** from the **Project** menu and SmartRisk Analyzer investigates your application's source code. An analysis takes anywhere from a few minutes to several hours depending on the size of your application and the speed and memory of the host system. Progress is posted in the **Results** window in the **Output** panel at the bottom of you screen.

## Run Security Scans

Upon completion, the **Binary Analysis** displays the application's source code in the main window. Run a **Full Security Analysis** from the **Scans** menu to receive a report on discovered security vulnerabilities. The scan is a short process and takes only a few moments. Once finished, SmartRisk Analyzer posts a summary in the main window panel. The results are listed graphically by severity and type. The **Annotation List** (at the bottom in the **Output Panel**) describes the results in further detail. Double-click any discovery to view the actual code. A sticky note for that discovery describes why the line is a security vulnerability and offers a suggested fix.

## Export to Reviewer Data

Export the results to a read-only version for development team distribution by saving the project as a **SmartRisk Analyzer Database Document**, or \*.srf file. Developers can install Reviewer versions of SmartRisk Analyzer on their






development systems to view the data SmartRisk Analyzer reports, add notes to the findings for instructions, delete findings, and increase or decrease the severity with stickynotes.

**Analyzing Output Data**

SmartRisk Analyzer finds vulnerabilities and offers suggestions on correcting security flaws in the original application. The severity of the flaws are categorized by their threat level according to the specific vulnerability. Developers can use this ranking to prioritize their security fixes according to their release schedules.

The five degrees of threat level severity are described below.

**Table 1: Five Levels of Severity**

Severity Warning	Description
<p><b>Severe Error</b>   Severe Error</p>	<p>SmartRisk Analyzer determines the code has serious security vulnerabilities and is an easy target for an attacker. You should modify the code immediately.</p>
<p><b>Error</b>   Error</p>	<p>The code has security vulnerabilities, and should be modified immediately.</p>
<p><b>Possible Error</b>   Possible Error</p>	<p>The code could be a security vulnerability, and become a target for an attacker. Review this code manually to determine if it contains a security vulnerability.</p>
<p><b>Warning Error</b>   Warning</p>	<p>The code might eventually result in a security vulnerability, but does not represent a high immediate risk.</p>
<p><b>Informational Alert</b>   Informational</p>	<p>Cross-references and informative messages.</p>

## Supported Platforms and Environments

SmartRisk Analyzer supports applications built in C on Windows and Solaris.

### Solaris Platforms

For Solaris 2.7 using GCC 2.95, compile using the following:

- The source code
- The binary executable (compiled with specific flags, as described below)
- Compile using `-gdwarf -g3` for maximum debugging information.

Do not compile with optimization, or use any `-O` options.

@stake does not recommend compiling applications to be analyzed by SmartRisk Analyzer under Solaris using the following:

- `-mflat`
- `-mno-faster-structs`
- `-mimpure-text`
- `-mcpu+{v9, ultrasparc or ultrasparc3}`
- `-mtune=v9, ultrasparc or ultrasparc3}`
- `-mlittle-endian`
- `-m64`
- `-mcmmodel`
- `-mstack-bias`

### Windows Platforms

#### Microsoft Visual C® and Visual Studio®

Under a **Microsoft Visual C** or **Visual Studio** environment, SmartRisk Analyzer requires the following objects:

- The source code
- The binary executable (compiled with specific flags, as described below)
- The “PDB” debug information file generated by the compiler

#### Microsoft Visual C++® 6.x

Using applications created by **Microsoft Visual C++ 6.x**, change the following project settings:

- In C/C++:
  - Remove the `/GZ` option from the options list.
  - In **Linker: Debug**, set **Generate Debug Info: Yes**

#### Microsoft Visual C++ 7.0 and Visual Studio .NET

For applications created using **Microsoft Visual C++ 7.0** and **Microsoft Visual Studio .NET**, change the following project settings:

- In C/C++ General:
  - Set **Debug Information Format** to **Program Database (/Zi)**.

- In C/C++ Code Generation:
  - Set **Basic Runtime Checks** to **Default**.
  - Set **Runtime Library** to **Single-threaded Debug**.
  - Set **Buffer Security Check** to **No**.
  - In **Linker: Debug**, set **Generate Debug Info**: **Yes**
- In Linker General:
  - Set **Enable Incremental Linking** to **No (/INCREMENTAL:NO)**.

**Microsoft Visual C++  
7.1 and Visual  
Studio .NET 2003**

For applications created in **Microsoft Visual C++ 7.1** and **Microsoft Visual Studio .NET 2003**, change the following project settings:

- In C/C++ General:
  - Set **Debug Information Format** to **Program Database (/Zi)**.
- In C/C++ Code Generation
  - Set **Basic Runtime Checks** to **Default**.
  - Set **Runtime Library** to **Single-threaded Debug**.
  - Set **Buffer Security Check** to **No**.
  - In **Linker: Debug**, set **Generate Debug Info**: **Yes**
- In Linker General:
  - Set **Enable Incremental Linking** to **No (/INCREMENTAL:NO)**.

## Deep Binary Modeling

In order to perform a deep binary analysis, SmartRisk Analyzer models the application, starting with the binary executable, since it is the most accurate, detailed, and complete representation of the application.

By performing deep binary analysis, SmartRisk Analyzer determines how the application performs in the given environment (from the operating system to the library files to the hardware), and also considers the transformations the compiler made when generating the machine code.

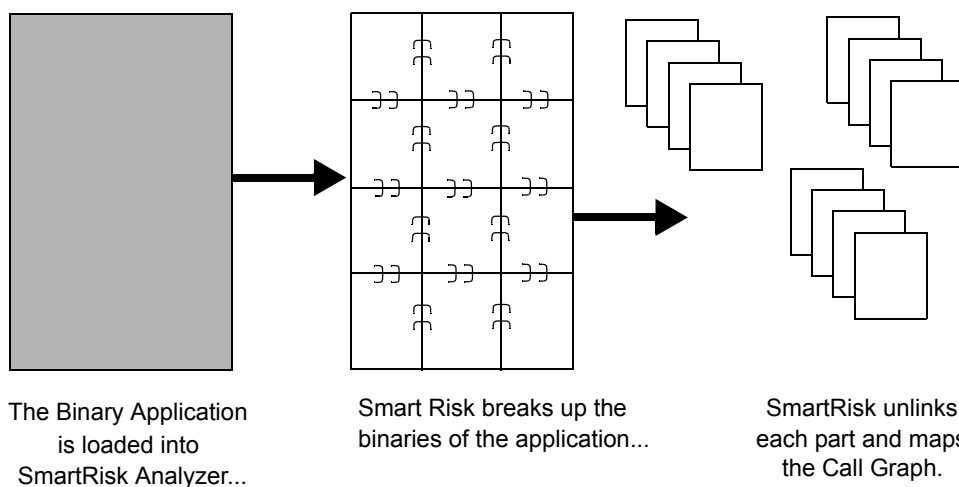
### Loading

SmartRisk Analyzer loads the application into a virtual machine memory, just as the operating system loads the program into real memory. Here, SmartRisk determines all of the interfaces the program uses through the dynamic link library files, such as API calls to the operating system, networking, or third party programs the application uses, such as a custom cryptography library.

### Unlinking

Once SmartRisk Analyzer loads the program, it breaks up the monolithic binaries into the individual functions originally created by the developers when writing the application. And instead of linking the parts together that make up the application like the compiler does, SmartRisk unlinks these parts to analyze each individual function.

The result is a call graph of the entire program with all the entry points. The SmartRisk Analyzer data flow modeler uses the call graph to map the interprocedural data flows within the program.



**Figure 2: SmartRisk Analyzer Loads the Binary Application for Modeling**

## Data Flow

The details of how data flows within an application is what potentially makes that application vulnerable to an attack. The SRA data flow graph connects variables to sources and destinations in the application, making the data flow graph an important dimension of an application model used for security analysis.

Attackers need specifically crafted data in order to exploit an application's security flaws. If attackers gained access to the network, they can manipulate code in applications with security vulnerabilities. SmartRisk Analyzer reads binary code to determine whether they are correct, risky, or a serious error.

Security scans use the data flow to perform risk analysis on potentially dangerous coding constructs. Analyzer compares the pieces of information to determine which parts of the application contain serious security vulnerabilities.

Even before the data flow is graphed, SmartRisk Analyzer discovers variables through a process called variablization. In variablization, SmartRisk Analyzer examines the assembly language instructions of the compiled program for a specific hardware platform. The instructions are analyzed as to how they manipulate the data, moving it between CPU registers and main memory.

From these instructions, rather than mapping variables to registers through register allocation like a compiler performs, the registers are mapped back into the variables the programmer specified and the types of these variables are discovered. For each procedure in the program, the input, output, and local variables are discovered and placed into the data flow graph.

Once the variables are discovered, the data flow is mapped as variables are initialized and data is moved into and out of them.

## Control Flow

SmartRisk Analyzer uses the control flow of the functions in the program to discover the high level language constructs the programmer used, and identify *if/then* statements, loops, and function calls. Later in the analysis, the security scans use the control flow graph to determine whether or not return values are checked before use. The control flow graph is an important input to the range propagation process.

The control flow graph is generated by the branching, jumping, and calling instructions contained in the assembly language for a function. The function is broken down into basic blocks, and how code executes in sequence, and the conditionals. From understanding the way the compiler uses conditionals to compile *if/then* statements and *loops*, Analyzer recreates a functionally equivalent high level representation of the control flow the way the programmers intended.

## Range Propagation

The range of a variable's potential value provides important information for security analysis. For example, the code below shows a conditional:

```
if (n < 100)
    strncpy(dest, src, n)
```

The range of **n** is between 0 and 99 (assuming **n** is an unsigned integer) within the basic block of true clause of the conditional. Security scans use this information to find vulnerabilities, such as buffer overflows, in code. The SRA security scans use the knowledge that the value of **n** must be lower than 99 to determine if the destination buffer, **dest**, is large enough to fit potential range of sizes of the source buffer, **src**.

## Analyzing the Application

SmartRisk Analyzer uses the detailed model that it builds to perform a security analysis on the application. The methodology of secure programming is to understand the limitations of the programming environment used to build and maintain an application. Programming within those boundaries contributes to secure code. Security scans determine if those boundaries have been exceeded and require tightening.

SmartRisk uses hundreds of different scans to identify potential flaws, determine whether those flaws are actual flaws, and then assign a risk level for each flaw. SmartRisk Analyzer is designed to minimize false positives, or pointing out correct pieces of code as potential flaws. The depth of its detailed binary model enables SmartRisk Analyzer to gather sufficient information on flaws to report an accurate analysis.

### Triggers and Analysis

The security scan starts by looking for trigger points within the code. Triggers are procedure calls that are known to be problematic, such as the **sprintf** family of functions in the C standard library or security critical operating system APIs.

Once a trigger point is located, the scan evaluates the context of the trigger in the data flow, the control flow, and the range propagation models. This is the “risk analysis” portion of the scan, which determines if the code actually contains a security flaw or not. If it is a flaw, then the severity is ranked. If there is not enough information in the application model to determine the severity, it is ranked as a **Possible or Informational Alert**, and provides the user with information about the trigger for manual analysis. The risk analysis may also not find a flaw within the trigger, and simply move to the next trigger. SmartRisk Analyzer's risk analysis functions use the depth of the binary application model to minimize manual analysis.

## Types of Scans

SmartRisk Analyzer contains hundreds of triggers and associated risk analysis functions in several categories:

- Stack/Heap buffer overruns
- Format string vulnerabilities
- Error return checking
- Integer overflows/underflows
- Threading/race conditions
- Cryptography
- Database
- Denial of Service
- Reliability Issues
- Input Validation
- Privilege Escalation
- Network Issues

These categories account for the majority of security flaws in software applications, and are root causes that lead to exploits, such as remote executions of arbitrary code, privilege escalation, and denial of service.

Scan categories for triggers are determined through public sources, as well as an internal knowledgebase @stake uses as secure coding guidelines. Included in this knowledgebase are subtle and tedious flaws that are difficult to determine.

Below is an example of a trigger and risk analysis in action during a Binary Analysis performed by SmartRisk Analyzer.

```
void execute_cgi(int client, const char *path, const char *method, const char
*query_string)
{

    char query_env[255];
    sprintf(query_env, "QUERY_STRING=%s", query_string);

return;
}
```

This trigger in this example is the **sprintf** function call that contains a **%s** format specifier with no precision specifier. An example of a precision specifier is:

```
%.100s
```

for maximum of 100 characters. Simply using **%s** without a value leaves the character length open-ended, and subsequently vulnerable to an attack.

The trigger causes SmartRisk Analyzer to probe deeper into the control flow and data flow graphs of the program to determine the risk level of the code.

A **sprintf** function can take up to three or more parameters:

- The target buffer
- The format string parameters
- A variable number of source buffers

In this example, there is only one source buffer. The **sprintf** function assumes an unlimited size target buffer. The precision specifier is an integer positioned in front of the **S** in the format string to specify maximum string length (**%.100s**). The **sprintf** function starts by copying the format string character by character. When it reaches a **%s**, **sprintf** copies the data in the first source buffer into its place. When there is no precision specifier, the **sprintf** performs a memory copy of the data from the source buffer to the target buffer. It only stops when it reaches a NULL terminator in the source buffer.

If the **source buffer** size plus the format string size (minus 2 characters for the **%s**) is larger than the **target buffer**, there is a **buffer overrun**. The risk analysis function must determine the size of the **source buffer** and **target buffer** by performing data flow analysis. Once it has these sizes, simple math computes whether or not there is a potential overrun.

The size of the target overrun is easy to determine. The variable, **query\_env**, is a **local static buffer** the size of 255. The **source buffer** is different. It is passed into the function as a pointer to a buffer as an input parameter, **query\_string**. To determine its length, SmartRisk Analyzer performs an interprocedural data flow analysis. SmartRisk looks for all the places in the code where the function containing this trigger is called and performs a data flow analysis to determine the size of the buffer passed into that function. In this particular case, there is one place where the function is called, at the end of the following function, **accept\_request()**.

```
void accept_request(int client)
{
    char buf[1024];
    int numchars;
    char method[255];
    char url[255];
    char path[512];
    size_t i, j;
    int cgi = 0;          /* becomes true if server decides this is a CGI
                          * program */
    char *query_string = NULL;

    numchars = get_line(client, buf, sizeof(buf));
    i = 0; j = 0;
    while (!ISspace(buf[j]) && (i < sizeof(method) - 1))
    {
        method[i] = buf[j];
```

```

    i++; j++;
}
method[i] = '\0';

i = 0;
while (ISspace(buf[j]) && (j < sizeof(buf)))
    j++;
while (!ISspace(buf[j]) && (i < sizeof(url) - 1) && (j < sizeof(buf)))
{
    url[i] = buf[j];
    i++; j++;
}
url[i] = '\0';

if (strcasecmp(method, "GET") == 0)
{
    query_string = url;
    while ((*query_string != '?') && (*query_string != '\0'))
        query_string++;
    if (*query_string == '?')
    {
        cgi = 1;
        *query_string = '\0';
        query_string++;
    }
}

    execute_cgi(client, path, method, query_string);

close(client);
}

```

The data flow graph continues up through the function call to `execute_cgi()`. The variable `query_string` is assigned from the variable, `url`, which is a local status buffer 255 in size. Now the source is determined. Returning the trigger source code, the size calculation using our buffer size can be performed to determine risk.

```
sprintf(query_env, "QUERY_STRING=%s", query_string);
```

The mathematical formula here is:

$$\text{Source length} + \text{format string} - 2 \text{ (for the \%s)} = 268.$$

Target length is 255.

The target buffer is not big enough. If an attacker crafts a query string with a length greater than 242, there will be a buffer overrun and an attacker manipulating the size of `url` could execute arbitrary code. This is considered a **Severe Error** and is marked as such by SmartRisk Analyzer.

The risk analysis function ranks each detected flaw into five levels of severity. Ranked from highest to lowest the levels are:

- Severe Error
- Error
- Possible Error
- Warning
- Informational

Note that in the code used for this example, the programmer did a good job of reading data from the network with a hard size limit, and then parsed the data safely into a few buffers. However, a security flaw arises when data is passed to another function and the subtlety of the `sprintf` function is involved.

## Reporting

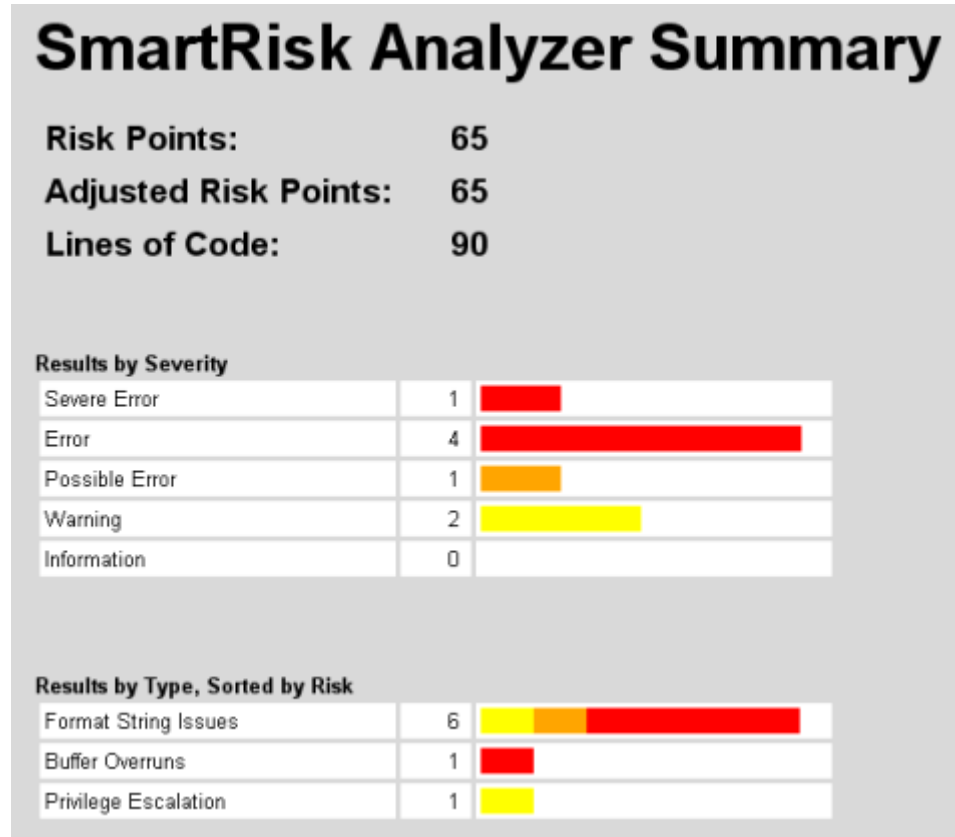
SmartRisk Analyzer produces two types of reports.

- **Summary Reports** are designed for QA or management who are tasked with tracking security quality from a project or organization perspective.
- **Detailed Reports** are designed for the developer who is assigned to remediate the security flaws.

### Summary Reports

Summary reports rank the security flaws to the **Adjusted Risk Points**. Adjusted Risk Points is the number of security flaws and their severity (each severity level has a value) divided by the size of the application (the total lines of code). Adjusted Risk Points gives a security quotient for the entire application, and a measurement of the overall security quality, used to track security remediation of an application or rating the acceptance test of an application. Higher Adjusted Risk Points in a Summary Report indicate riskier applications with many vulnerabilities.

Adjusted Risk Points can also be used to compare the relative risk of the different applications developed and used within an organization. The summary report also contains charts of the security analysis **Results by Severity and by Type, Sorted by Risk**. A customer can run SRA at various application milestones and compare the Adjusted Risk Point values over time to get a feel for if code security is improving.



**Figure 3: Summary Report Screen**

### Detailed Report

A Detailed Report is the core output. This information points to the exact code where the security flaw resides, details the security error in the code, and suggests a fix. In the `printf()` example given previously, SmartRisk Analyzer discovers the code is a potential buffer overflow error, and recommends to use a precision specifier to limit the size of the source buffer copied to the target buffer.

SmartRisk Analyzer presents its reports in a highly interactive user interface very similar to a modern IDE (integrated development environment), so that SmartRisk Analyzer is familiar to developers. The **Annotation List** of details are in a Results window at the bottom of the screen with sortable columns for severity, ID, flaw location, code title of the flaw, and explanation/remediation help. Sorting on the source code location column enables viewing the issues contained in a particular source code file for assignment to the appropriate programmer.

Severity	ID	Location	Title	Description
Severe Error	2	finger.c.markup, Line 32 (id=35676)	strcpy	This call to strcpy() appears to contain a buffer o
Error	7	finger.c.markup, Line 68 (id=43855)	sprintf	This call to sprintf() contains a potential buffer ov
Error	5	finger.c.markup, Line 47 (id=44649)	sprintf	This call to sprintf() contains a potential buffer ov
Error	4	finger.c.markup, Line 31 (id=35761)	scanf	This call to scanf() contains a potential buffer ov
Error	3	finger.c.markup, Line 27 (id=35734)	scanf	This call to scanf() contains a potential buffer ov
Possible Error	6	finger.c.markup, Line 59 (id=40616)	sprintf	Be aware of format string errors when calling spr
Warning	8	finger.c.markup, Line 45 (id=39158)	gethostby...	DNS results can easily be forged by an attacker
Warning	1	finger.c.markup, Line 89 (id=46003)	fprintf	Calling fprintf() with a variable format string can l


Results  Annotation List

Figure 4: Detailed Summary

Double-click a flaw in the **Annotation List**, and SmartRisk jumps to the selected source code for that flaw in the main window.

```

printf("\nuser: ");
scanf("%s", &inbuf);
strcpy(user, inbuf);

fd=socket(AF_INET, SOCK_STREAM, 0);
if (fd== -1){
    perror("socket");
    exit(1);
}

```

The source code is annotated with an icon indicating the severity of the issue. For more on **SmartRisk Severity Levels**, please refer to [page 2-4](#).

**Click the severity icon** for a sticky note with an explanation of the issue and remediation help. These notes give the programmer information regarding the code under inspection, why it is a flaw, and how to fix it.

```

[-] Severe Error(finger.c.markup, Line 32 (id=35676)): strcpy
This call to strcpy() appears to contain a buffer overflow
error. The source buffer appears to be 100 bytes, and the
destination buffer is 50 bytes. This type of error can
directly result in a security vulnerability and should be
fixed immediately.

```

These annotations are not static, and can be **edited**, **created**, and **deleted**. Modifying annotations is useful, particularly when analyzing applications containing complex data manipulation that is hard to reach with an automated analysis. SRA cannot determine the severity in some complex applications, and as

such SmartRisk ranks the code as a **Possible Error**. A Possible Error contains all the available information, and SmartRisk describes how to manually determine if the questionable code is an error or not. Check the code to make a manual determination, and modify the **Possible Error** annotation to a new severity or deletes it.

## Workflow

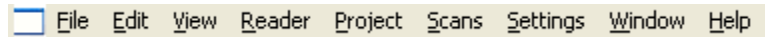
SmartRisk Analyzer fits the workflow of either a centralized application security group responsible for the security of many applications across an organization or a software development team building a single application. The workflow consists of five steps:

- Load full source code and compiled binary
- Run the Binary Analysis
- Run Security Scans
- Export to Review Data (the SRA Reviewer)
- Analyzing Output Data

For more details of these five steps, please refer to [page 2-3](#).

## User Interface Menu Options

SmartRisk Analyzer is streamlined with a Visual Studio-style interface, and Windows-type menu options that should be familiar.



- File** The **File** menu runs typical Windows file menu commands. Unique to SmartRisk Analyzer are options to **Save**, **Open**, and **Close** SRA projects.
- Edit** The **Edit** menu enables typical Windows edit commands, such as **Cut** and **Paste**, and **Find** and **Replace** text.
- View** The **View** menu opens and closes displays in SmartRisk Analyzer.
- **Workspace** - Located on the left hand side of the SmartRisk Analyzer screen and displays the structure tree of you project.
  - **Output** - The Output panel is at the bottom of the SmartRisk Analyzer screen and displays **Results** and the **Annotation List**.
  - **Annotation List** - A Tab from the Output panel, this lists the found security vulnerabilities.
  - **Analysis Summary** - The large panel to the right of the screen displays the findings graphically. Another tab displays the original code that is analyzed.
  - **Immediate Window** - Located in the Output panel.
  - **Process Memory** - A screen in the main window with memory locations.
  - **Code Browser** - Shows hexadecimal locations in the Workspace.

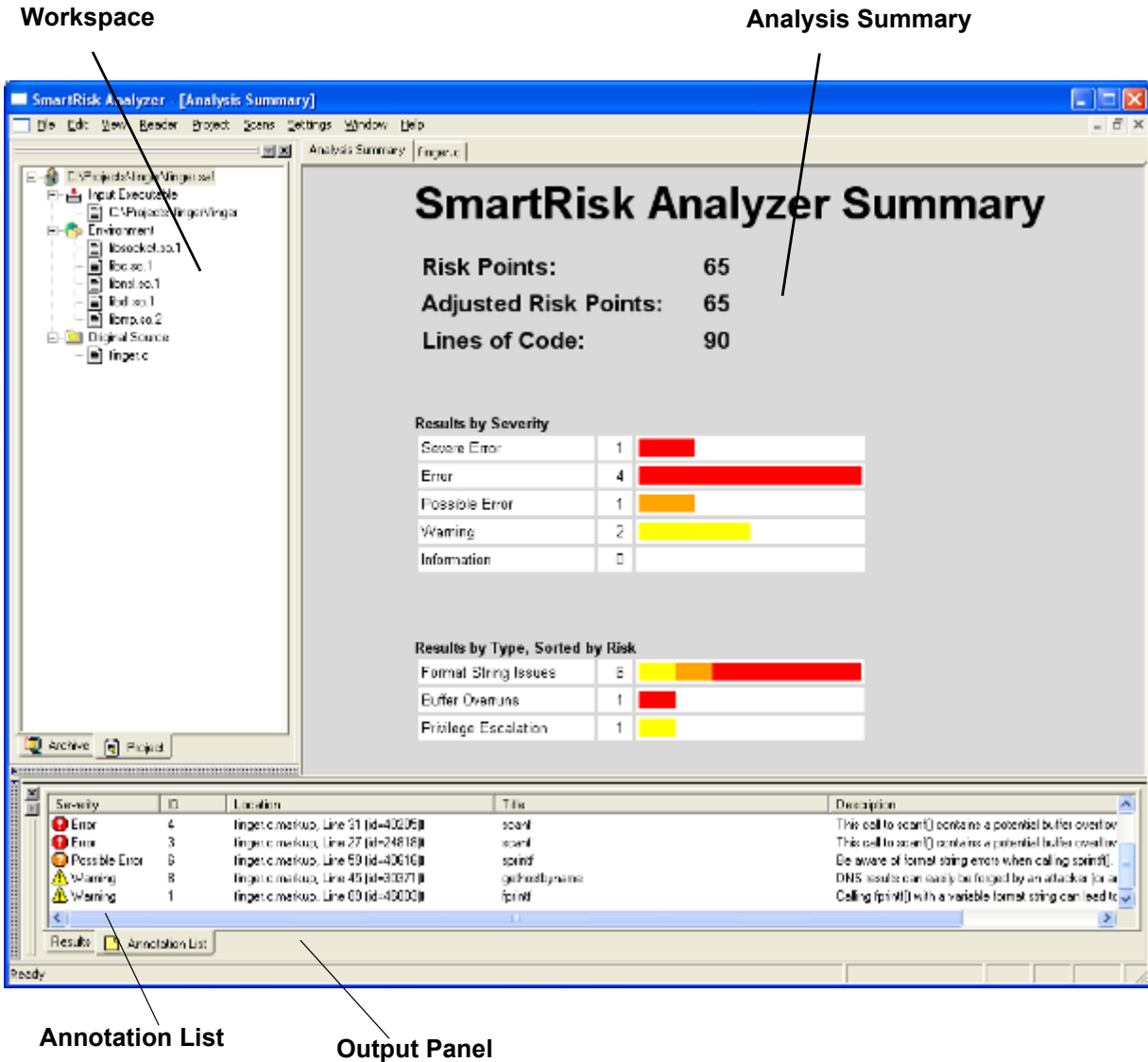


Figure 5: SmartRisk Analyzer Full Screen Shot

**Reader**

- **Generate Reader Document** - After Binary Analysis and a full security scan completes, generate a reviewer document to distribute to the development team. Reviewer documents read-only filesm, and are viewed using both the Reviewer version and full version of Analyzer. Review documents are also smaller in size than the full SmartRisk Analyzer project files.
- **Reader Document Settings** - Settings for reader document.

**Project**

The Project menu provides the following functions:

- **Starts Binary Analysis (F5)** - Perform this function after you start your project.

- **Stops Binary Analysis** - Terminates the Binary Analysis before completion.
- **Resets Project** - Clears the project's current analyzed state.
- **Settings** - A dialog window pops up to change the **Binary Analysis Settings** that are set up when creating the new project, such as the **Input File** and **Default Compiler**. You have the option of changing the **Environment** setting the application is created, as well as the compiler settings.

The **Advanced** tab sets specific functions during the Binary Analysis, such as:

- Force input table/PLT/GOT read-only after binding
- Do not unlink
- Ignore debugging symbols
- Do not analyze library calls

**Note:** Changing these settings after a Binary Analysis will affect your current project and most likely result in a program error.

## Scans

- **Global Scans**
  - **FullSecurityAnalysis** - Perform a **Full Security Analysis** after running the **Binary Analysis**. The scan searches the application for specific vulnerabilities and summarizes the findings upon completion.
- **Stop Running Scans** - Terminate the scans before completion.

## Settings

There are two options under the **Settings** menu.

- **Options** enables or disables the wizard dialog box to appear at startup.
- **License Key** displays your Machine ID and the License Key for your running version of SmartRisk analyzer.
- **Personality Settings** adds additional users to the SmartRisk Analyzer. You may also customize the sticky notes for each user.

## Window

You can organize SmartRisk Analyzer windows in desirable order and arrangement.

## Help

Select **Help** to launch the **SmartRisk Analyzer User's Guide**.

## Right-Click Menu

Select original source code in the main panel and right-click for additional menu options. You may add **stickynotes** or run **Local Scans** in addition to typical Windows right-click options.



# CHAPTER 3: INSTALLATION

Before you install SmartRisk Analyzer, make sure you have you a SmartRisk Analyzer Installation CD, all programs are closed, and that your target system matches the minimum system requirements outlined in *Chapter 1: Getting Started*.

**Note:** You are required to contact @stake via email or telephone for an **Activation Key**. @stake replies to Activation Key inquiries during regular business hours, 9 AM – 5 PM, Monday – Friday EST.

## Installing SmartRisk Analyzer

The installation instructions below assume your system meets the minimum requirements detailed on *page 1-2* to install and run SmartRisk Analyzer.

1. Close all Windows programs.
2. Insert your @stake SmartRisk Analyzer Installation CD into your main CD-ROM or DVD-ROM drive.
3. Open the CD through Windows Explorer.
4. Drag the SmartRisk Analyzer directory to your hard drive under the root directory (i.e. C:\SmartRisk).
5. Open the directory, double-click the **SmartRisk.exe** file to open SmartRisk Analyzer.
6. You are prompted to enter an **Activation Key**. Send the **Machine ID** information from the Activation Key Window to [support@atstake.com](mailto:support@atstake.com) or call 1.866.621.3500 for a Key. @stake responds to Activation Key requests during regular business hours, 9 AM to 5 PM, Monday – Friday EST.

**Note:** You must enter the key unique to your Machine ID. The Machine ID codes are generated from pieces of information on your system, and one key will not work with all systems. If you need to move SmartRisk Analyzer to another system or you need to rebuild your system, you will need to contact @stake again for a new key.

7. Once the **Activation Key** is added, SmartRisk Analyzer is ready to analyze your source code. For a walk-through tutorial, please refer to *Chapter 4*.

## Uninstalling SmartRisk Analyzer

To uninstall SmartRisk Analyzer, follow the steps below.

1. Open Windows Explorer, and navigate to the directory containing SmartRisk Analyzer.
2. Delete the directory from your hard drive.
3. Empty the Recycle Bin.
4. The program is now removed from your system. To re-install SmartRisk Analyzer on another system, you will need to request a new Activation Key unique to that system.

# CHAPTER 4: SAMPLE PROJECT WALK-THROUGH

This chapter walks you through a typical SmartRisk Analyzer project using a sample application provided to you during installation. The sample project contains commonly found security vulnerabilities in applications.

The walk-through is intended to familiarize developers with the procedures and actions of SmartRisk Analyzer, point out common security vulnerabilities, and usability of SmartRisk Analyzer.

A typical analysis involves five steps of operation:

- Load full source code and compiled binary
- Run the Binary Analysis
- Run Security Scans
- Generate Database Documentation (the SRA Reviewer document)
- Analyzing Output Data

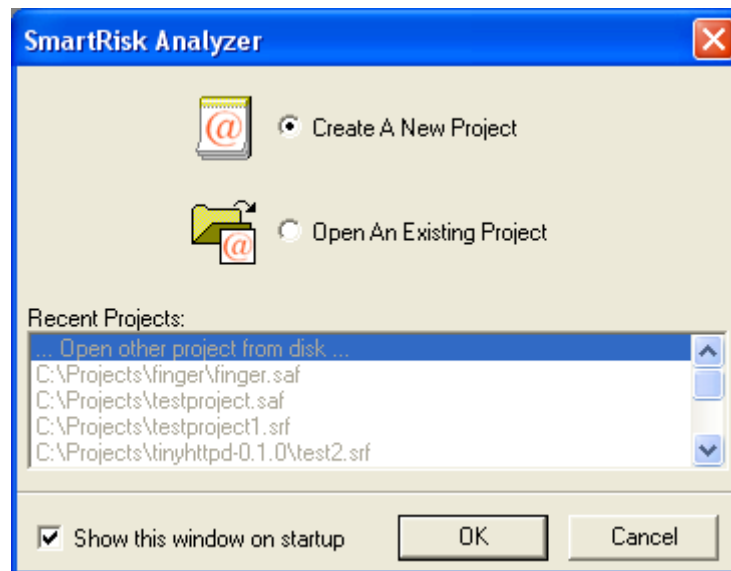
## Walk-Through Using SmartRisk Analyzer

Follow the instructions below to walk-through a sample project. The sample project is copied on to your system during installation of SmartRisk Analyzer.

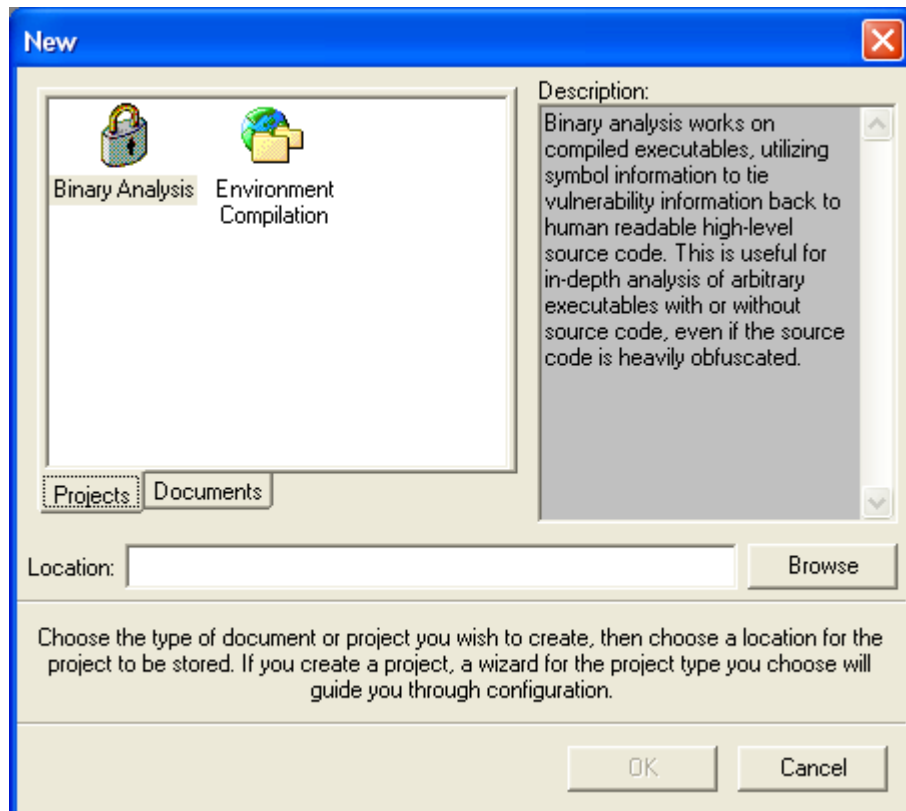
**Note:** You may use this walk-through to guide you through one of your own applications.

1. Run SmartRisk Analyzer.

2. A dialog window offers a choice to **Create A New Project** or **Open an Existing Project**. Choose to Create and then click **OK**.



3. A dialog box offers you to open a **New Binary Analysis** or **Environment Compilation**. Click on **Binary Analysis**, then click **Browse** for an **Open** dialog box.



4. An open window requests you to **Choose Project Location**. Browse to:

C:\SmartRisk\sample applications\finger

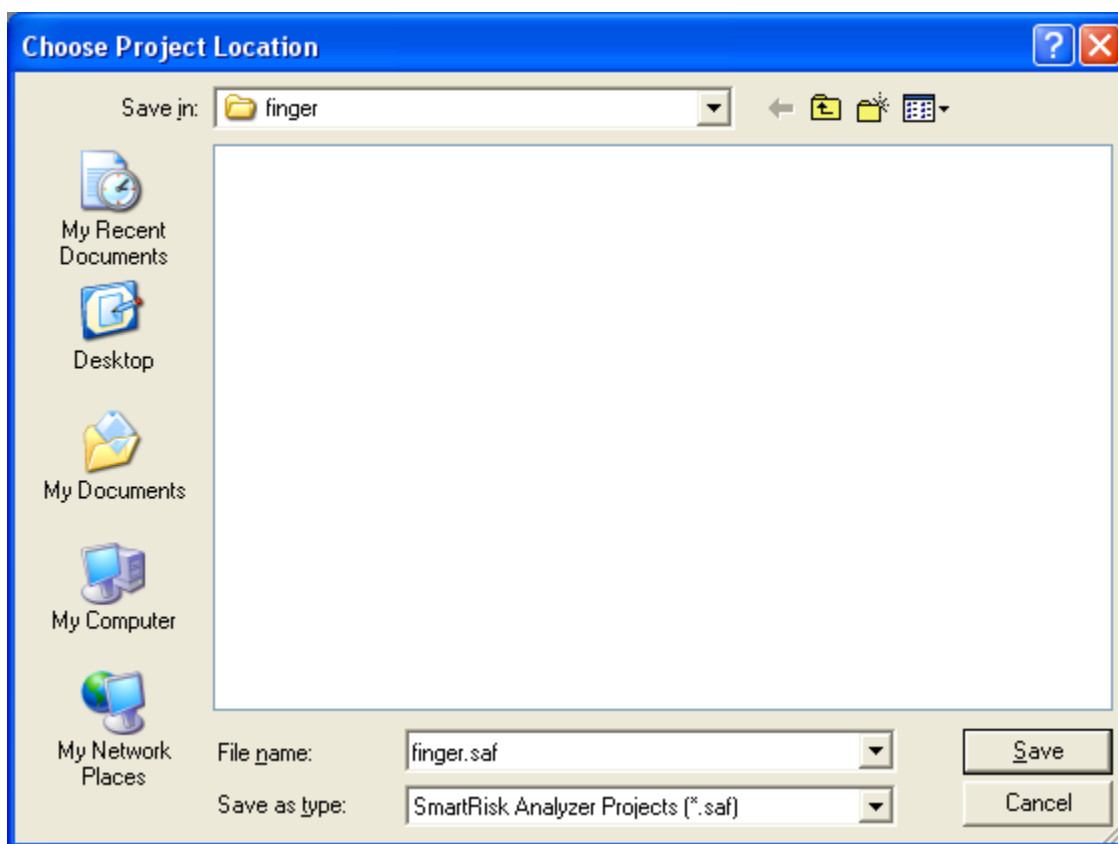
and name your project **finger.saf**. Then click **Save**. If a project already exists by that name, overwrite the project or create another called **finger1.saf**.

Your SmartRisk project file (\*.saf) *must* be saved and stored in the top level directory for the application in which you are analyzing. For example, if you project has the following directory structure:

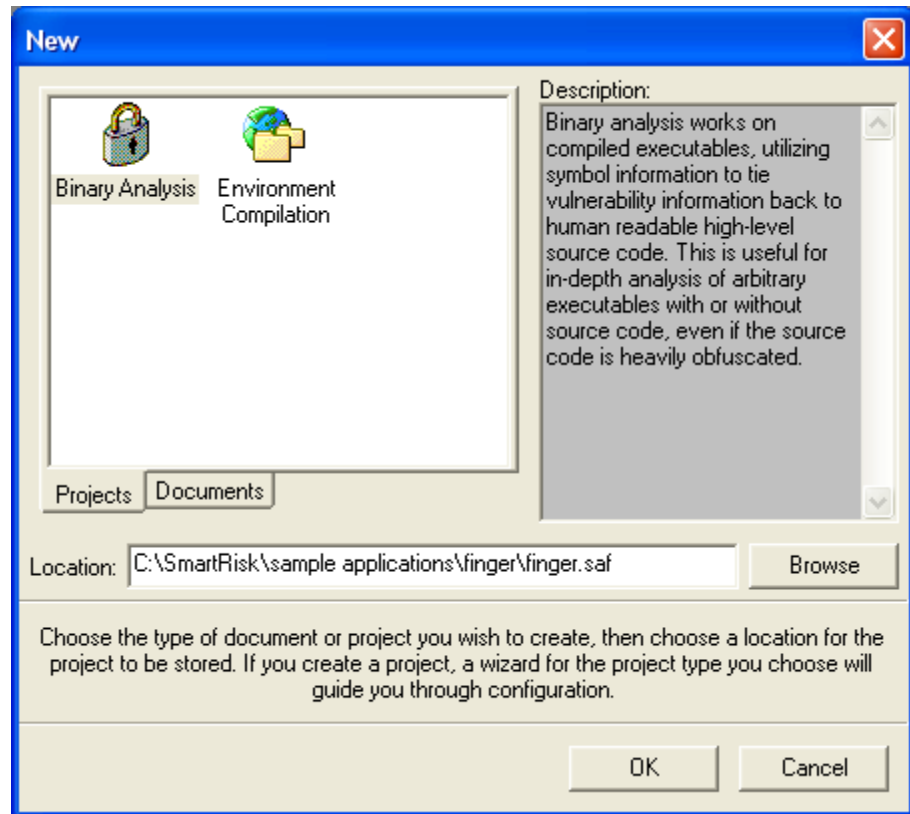
```
\ntp
\ntp\ntpd
\ntp\libntpd
```

and you are analyzing the binary file `\ntp\ntpd\ntpd`, which used a library source in `\ntp\libntpd`, you would need to put your \*.saf project file in `\ntp`.

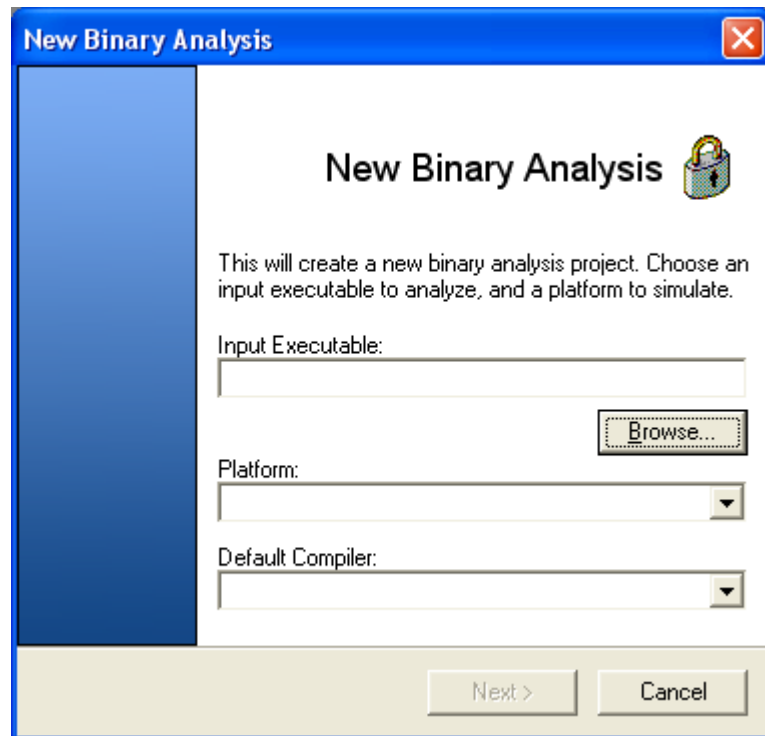
SmartRisk Analyzer conducts searches downwards through the directory structure to find the required libraries and source files.



5. You return to the **New** dialog, now with address information in the **Location** window. Click **OK** to continue.



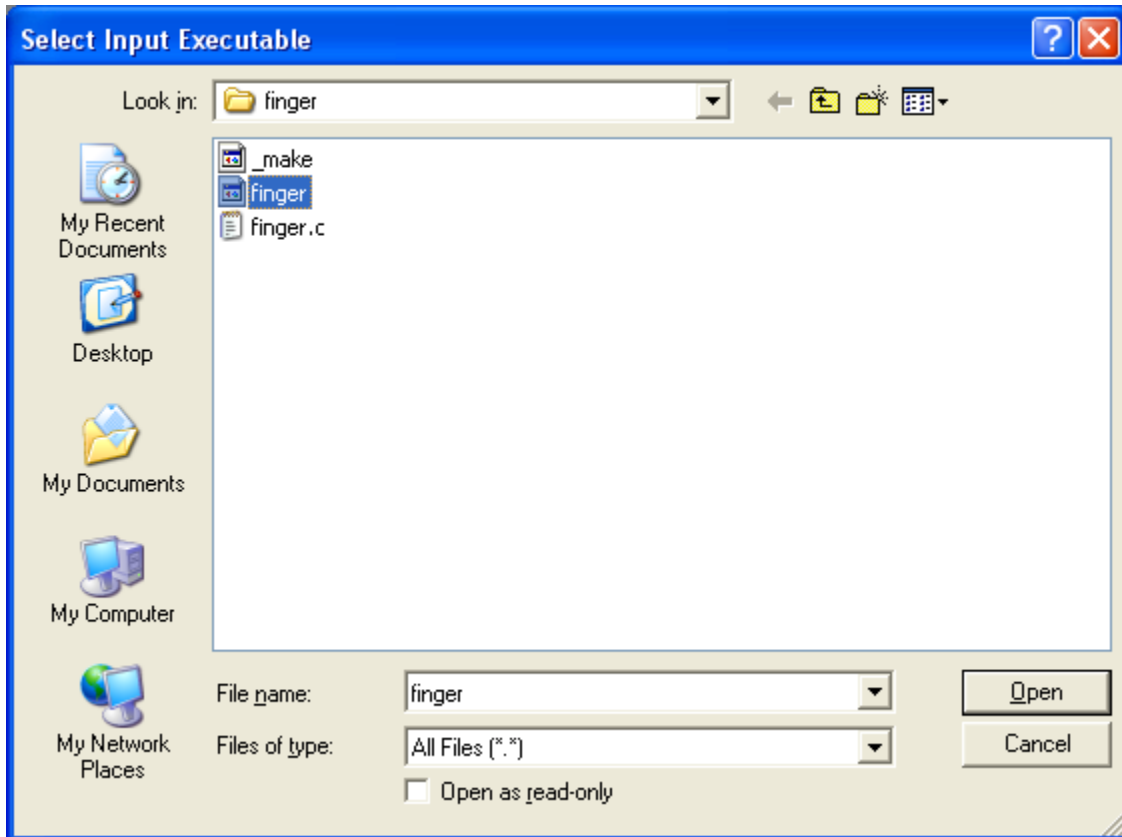
6. Now create a **New Binary Analysis**.



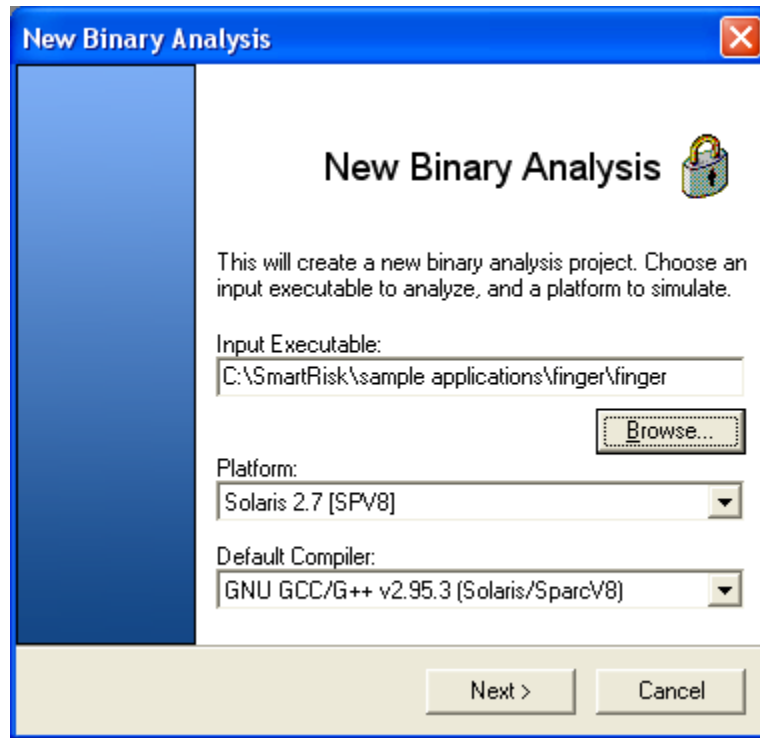
Click the **Browse** button to locate an **Input Executable**. This file is your actual program file.

7. Select an **Input Executable** in the directory:

C:\SmartRisk\sample applications\finger  
and select **finger**. Click to **Open**.

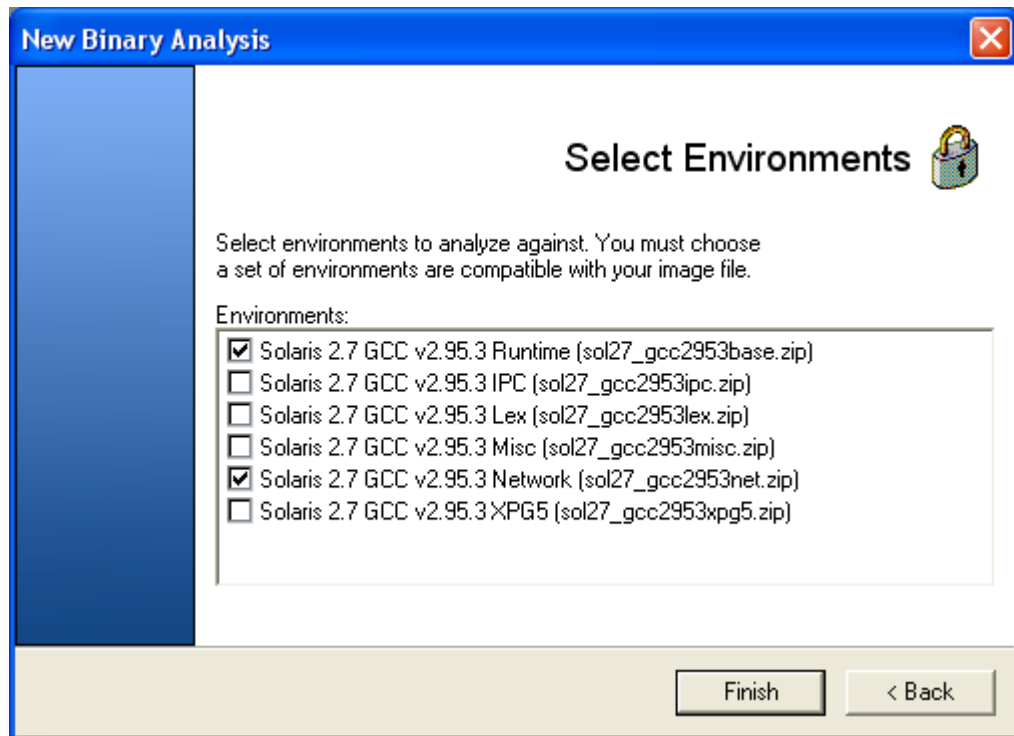


8. The **New Binary Analysis** dialog box fills in the **Input Executable**, **Platform**, and **Default Compiler** information for you.



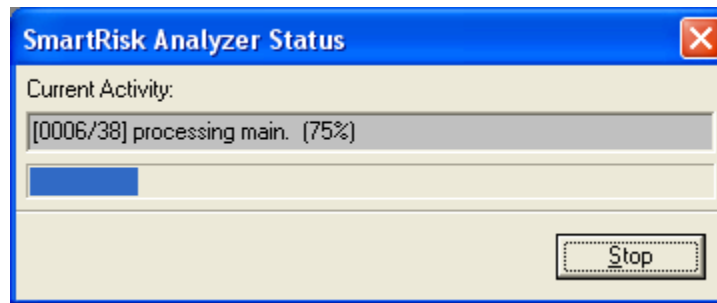
SmartRisk Analyzer opens and reads the binary file to verify support for the platform and compiler, and then fills in those options. You can change the **Platform** and **Compiler** information using pull down menus, however, alternative choices may cause an error. When you are ready, click **Next**.

9. A **Select Environment** window appears for compiler environment selection. SmartRisk Analyzer selects default choices read from your application. The list of choices should coincide with your previously selected platform. If you need to revise your previous choices, click **Back**. Click **Finish** when ready.

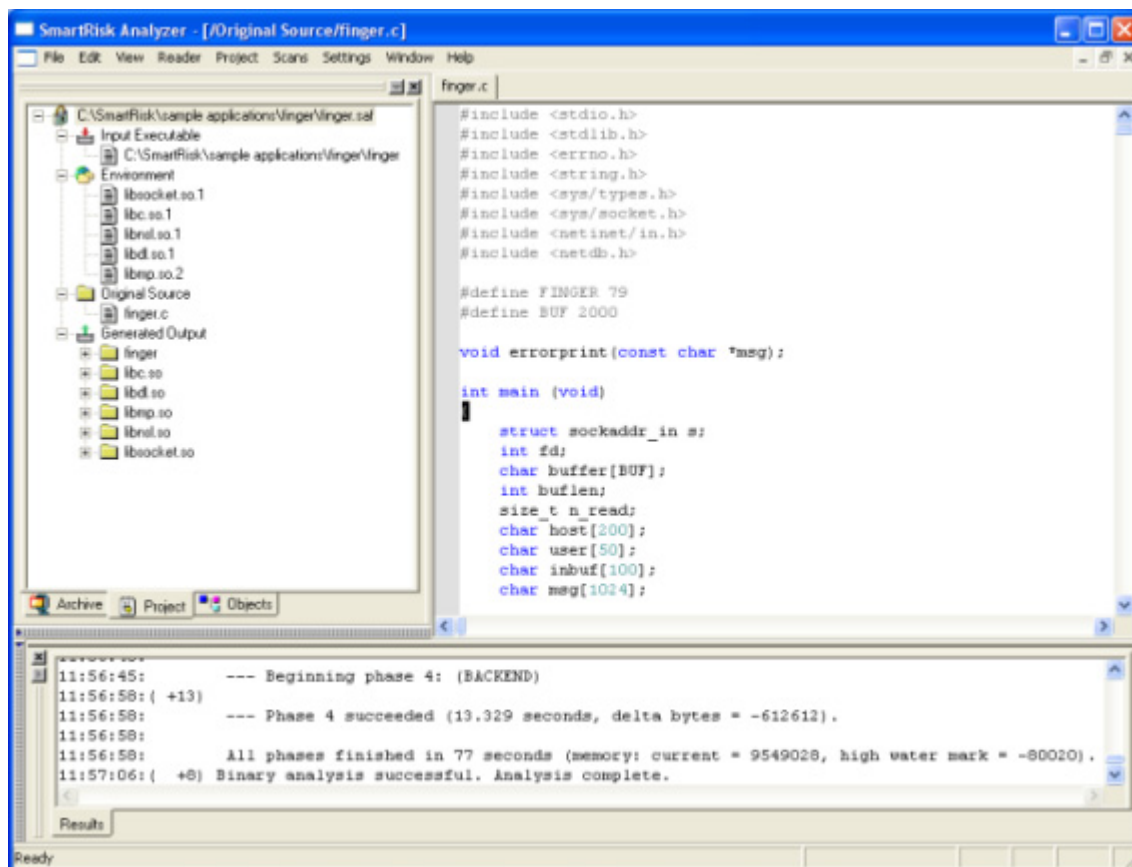


**Note:** Deselecting the default choices could result in a poor analysis. Selecting additional environments does not harm the analysis, however, it also does not ensure a better analysis.

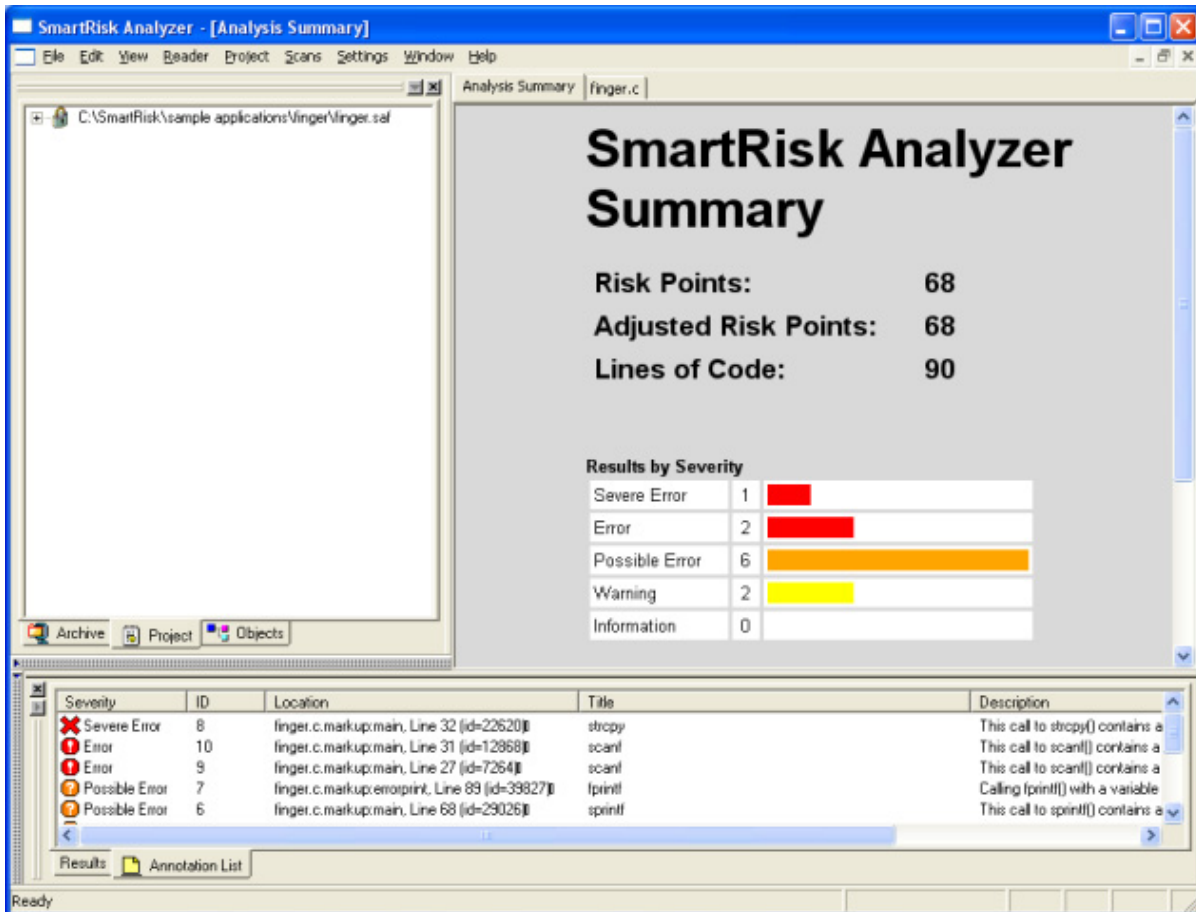
10. Your new project is ready for a **Binary Analysis**. This enables SmartRisk Analyzer to analyze the application and support files.  
Choose the **Project** menu to **Start Binary Analysis** or press **F5** to start. Binary Analysis takes anywhere from a few minutes to several hours depending on the size of your executable, the speed of your system's processor, and the amount of memory in your system. For more details on the Binary Analysis, see [page 2-7](#).



- Once **Binary Analysis** is complete, a copy of the source code appears in the main panel on the right-hand side. Use the **Scans** menu to select **Global Scans** then **FullSecurityAnalysis** to scan your application for security flaws.



12. The scan may take a few moments depending on the size of your application and system. When complete, an **Analysis Summary** of the results displays in the main panel. To view the code, click the tab at the top of the main panel.



13. An annotated list of found security flaws appears in the bottom panel. Double-click any of them to view the line of code.

```

printf("\nuser: ");
scanf("%s", &inbuf);
strcpy(user, inbuf);

fd=socket(AF_INET, SOCK_STREAM, 0);
if (fd== -1){
    perror("socket");
    exit(1);
}

```






14. Save the project under the **File** menu to **Save Project As** to your desired location when finished.

## Determining Severity Differences

Each security vulnerability SmartRisk Analyzer finds is tagged with a severity level. The level is determined by the vulnerability of the code as a target for an attack. The more vulnerable the code, the easier the attack and the higher severity that code is categorized.

As previously discussed in [Chapter 2](#), there are five categories of severity.

**Table 1: Five Levels of Severity**

Severity Warning	Description
<b>Severe Error</b>  Severe Error	SmartRisk Analyzer determines the code has serious security vulnerabilities and is an easy target for an attacker. You should modify the code immediately.
<b>Error</b>  Error	The code has potential security vulnerabilities, and the code should be modified immediately.
<b>Possible Error</b>  Possible Error	The code could create a security vulnerability, and could be a target for an attacker. You should investigate this code to determine if it contains a security vulnerability.
<b>Warning Error</b>  Warning	The code might eventually result in a security vulnerability, but does not represent a high immediate risk. It should be investigated further, because it could potentially contain vulnerable code.
<b>Informational Alert</b>  Informational	Cross-references and informative messages.

These levels of severity help developers determine which vulnerabilities take priority in fixing.

For example, SmartRisk Analyzer determines code is a **Severe Error**, and you should fix this immediately. The following is considered a Severe Error:

```
char user[50];
char inbuf[100];
....
strcpy(user, inbuf);
```

Here, the code contains one source buffer (**inbuf**) that is larger than the destination buffer (**user**). This is called a buffer overflow, and an attacker could use a buffer overflow to overwrite the application, and ultimately take over the network. To remedy the security flaw, the source and destination should have the same value, or at the very least a buffer limit should be imposed with a bounded string copy.

Coding like this example is common in applications where developers are under heavy schedules to complete the application for release. And while the application may functionally work, too often simple security flaws like buffer

overflows are overlooked in terms of security, and the application goes to the customer with unknown security vulnerabilities.

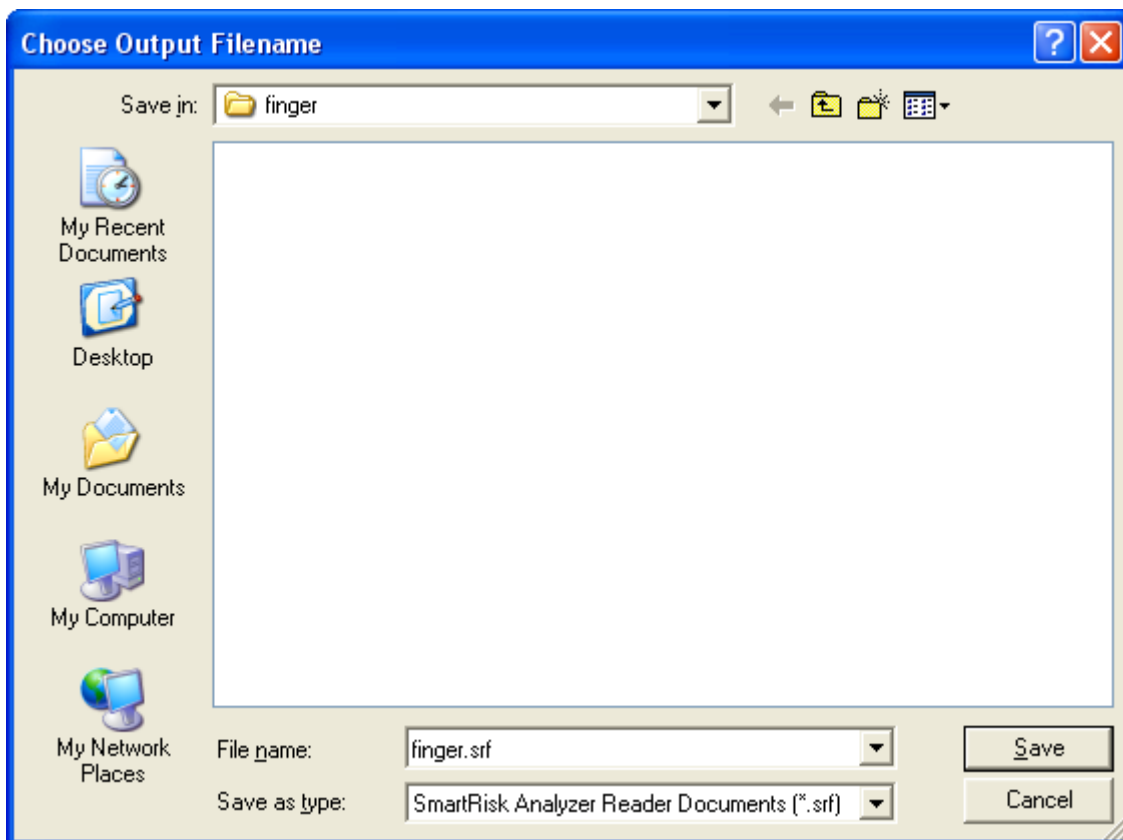
A severe error is more important than code that is labeled an **Informational Alert**. The code labeled Informational may not be a security vulnerability, but is cross-referenced to a more serious error. Informational Alerts can also provide additional information to a fix for a more serious flaw.

Since the information for known vulnerabilities is pointed out, it is advantageous to fix all the points, and limit malicious attacks to your application.

## Generating Reviewer Files

Once you have finished the Binary Analysis and Security Scan, the project can be saved as a **SmartRisk Analyzer Database Document** read-only file (\*.srf) for wide distribution to other developers who are running the SRA Reviewer. Reviewer documents are read-only formats, however, users can add notes to the project files. Users can not run scans on reader files or on the Reviewer version of SmartRisk.

To generate the review document, under the **Reader** menu, select **Generate Reader Document**. A **Save As** dialog box prompts you to **Choose your Output Filename** and location. Choose the name of the file and click **Save**.



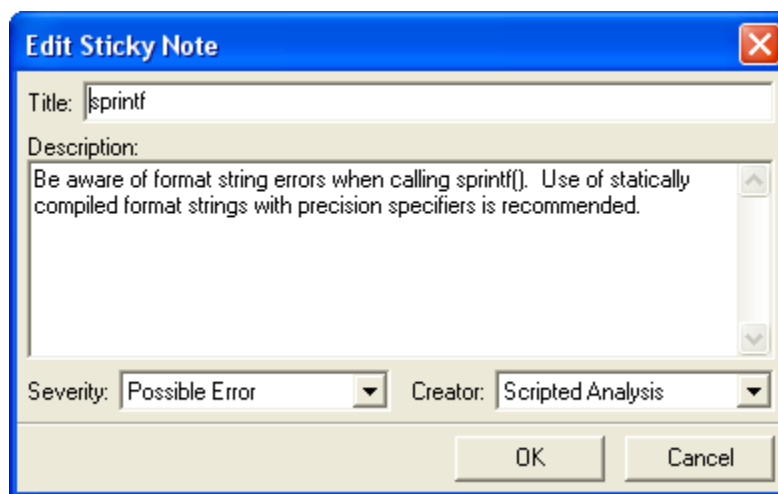
Once you have generated a database document, it can be distributed to the development team for assignments and review.

## Making Notes in SmartRisk Analyzer

Before generating a reviewer document for your development team, you may want to make notes on one or several of the discovered vulnerabilities. SmartRisk Analyzer uses sticky notes to describe found security vulnerabilities. By right-clicking the severity icon in the **Annotation List**, you can **Edit Note** or **Remove Selected Note**.

**Note:** **Remove Selected Note** deletes the listed item permanently.

Right-click any item in the **Annotation List** and select **Edit Note**. You must edit notes in the **Edit Sticky Note** dialog. You cannot edit the actual notes themselves.

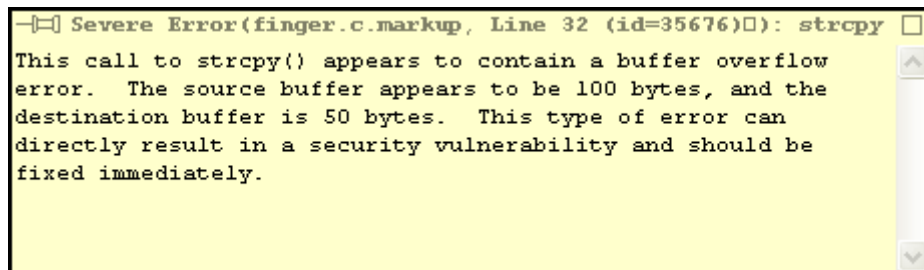


Each part of the note can be modified. If you would like to change the severity of the note, use the pull-down menu and select a different severity. SmartRisk Analyzer changes the severity icon in the Annotation List. You can also specify the reviewer by changing the **Creator** personality in the **Creator** pull-down menu in the **Edit Sticky Note** dialog. For more on adding creators, please see [page 4-14](#).

You may also want to add notes to code that was not found to have a vulnerability. Simply hover your mouse over the code until it highlights blue. Then right-click and choose to **Add a Sticky Note** to the line. You need to select a severity level. The default is **Informational**. Click **OK** to post the note.

## Viewing a Created Sticky Note

SmartRisk Analyzer automatically adds notes to all found vulnerabilities. These notes detail the discovered vulnerability and suggests a fix. Double-click the vulnerability listed in the **Annotation List**; that will bring you to the vulnerable code in a copy of the original source. Click the severity icon on the left hand side to view the sticky note.



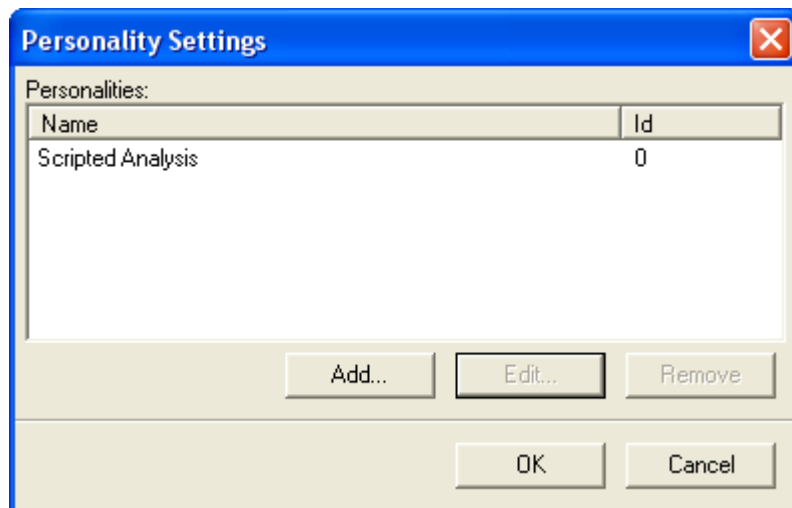
Any additional comments you make to existing notes will appear here as well.

To view the notes during another session in Smart Risk Analyzer, be sure to view the **Annotation List** by clicking the **View** menu and choosing **Annotation List**.

**Note:** If you run another scan using a saved Project file, any notes you have made or modified will be deleted.

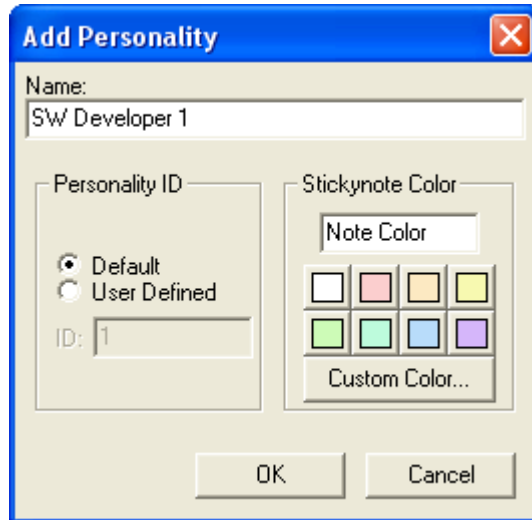
## Adding Creators

More than one developer may edit the same SmartRisk Project or Reviewer file. To add creators, click the **Settings** menu to view the **Personality Settings**. The **Personality Settings** dialog window pops up and the default user, **Scripted Analysis**, should already be listed with **Id 0**.



Click **Add**. The **Add Personality** window appears.

Choose a creator name. Every user has a unique ID. You may continue to have SmartRisk Analyzer assign the IDs or custom define it. Each user may choose their own sticky note color for easy identification.



Users can be edited or removed at any time.



# CHAPTER 5: PERFORMING SECURITY ANALYSES

After SmartRisk Analyzer runs a Binary Analysis on your application, and discovers security vulnerabilities, the discovered security flaws in your application can be fixed using the recommendations from SRA. This chapter discusses how to perform security analyses, and how to interpret the results.

## Analyzing an Application

SmartRisk Analyzer is an automated tool, and hence performs the tasks of binary analysis and security scans with minimal work on the part of the user. However, SmartRisk only analyzes complete applications, and therefore you should prepare the following prior to analyzing your application with SmartRisk Analyzer:

- The *complete* source code for the executable
- The binary executable and *all* required dynamic libraries
- Full debugging information for the application

SmartRisk Analyzer cannot analyze your application's security with all of these components.

## Creating a SmartRisk Project File

When SmartRisk Analyzer is executed, a wizard dialog box asks whether you would like to **Create a New Project**, or open an existing one. When you create a new project, you are loading your application into SmartRisk and saving your settings as a SmartRisk project file (\*.saf). The saved project file contains the information gathered during binary analysis and the security scan.

Your SmartRisk project file (\*.saf) *must* be saved and stored in the top level directory for the application in which you are analyzing. For example, if you project has the following directory structure:

```
\ntp
 \ntp\ntpd
 \ntp\libntpd
```

and you are analyzing the binary file `\ntp\ntpd\ntpd`, which used a library source in `\ntp\libntpd`, you would need to put your \*.saf project file in `\ntp`.

SmartRisk Analyzer conducts searches downwards through the directory structure to find the required libraries and source files.

### Loading the executable

After naming the project file, load the executable for your application. This must be the *complete* compiled binary executable in your application, as well as all the source code for your application, including third party applications in one central directory. In prepping the compiled binary, be certain to turn on all debugging information in your compiler.

SmartRisk asks you to name your new SmartRisk project, and save it to disk. Once that is done, locate the binary executable for SmartRisk to load. Now SmartRisk is ready for platform and compiler information.

### Choosing Platform and Compiler Options

SmartRisk supports the latest versions of Windows and Solaris operating systems, applications, and compilers. You should compile your application with the latest version, if you are not already doing so.

For more information on how to prepare executables in your compiler for an analysis in SmartRisk, please refer to [page 2-5](#).

Once the executable loads, SRA opens and reads the binary file to gather pieces of information about your application. One of the pieces of information SRA discovers is the platform and the compiler used to create the application. If SRA supports these, the binary loads, otherwise SRA returns an error.

SmartRisk automatically detects the platform and compiler information into the binary analysis project. SRA allows changes to these settings, however, changing them may result in an error.

### Choosing Environment Files

The environment files of your application help the binary analysis determine the type of application model to build in order to properly analyze for security flaws. SmartRisk Analyzer sets default settings according to the compiled build of your application.

SmartRisk determines which code was written by you or your developers, and which might be default code written by the compiler's manufacturer (i.e. Visual C++ applications contain default code written by Microsoft). SmartRisk recognizes that you want to run an analysis on code that you wrote, and blocks out the extra code to limit the workload. You may select environments that were not selected by default by SmartRisk, however, de-selecting options will result in an error. It is best to accept the default choices and run the Binary Analysis using those, rather than not running them.

**Note:** These environments must be compatible to your application.

## Binary Modeling

SmartRisk Analyzer builds a binary model of the application to analyze each individual function. In doing so, SmartRisk is able to read the data and control flows, and range propagation to find security flaws. Binary modeling decreases the chances of false positives in its analysis.

In order to properly model the binary files, the application must be loaded into SmartRisk as a completely compiled binary version of the application. Once loaded, SmartRisk begins to unlink the monolithic binary files into their individual functions. These functions are analyzed, and the full control flow, data flow, and range propagation information is mapped out in SRA's binary model.

## Security Scans

After binary analysis, SmartRisk runs a security scan of the application. It searches for certain triggers in code that could result in a security flaw. From there, it determines whether the trigger results in a security flaw or not.

SmartRisk Analyzer classifies the types of scans into the following categories:

- Stack/Heap buffer overruns
- Format string vulnerabilities
- Error return checking
- Integer overflows/underflows
- Threading/race conditions
- Cryptography
- Database
- Denial of Service
- Reliability Issues
- Input Validation
- Privilege Escalation
- Network Issues

The results of the security scans are posted in two reports: a **Summary Report** and a **Detailed Report**, both which point out discovered security flaws in the application, describe the problems, and suggest a security solution.

### Stack/Heap buffer overruns

Format strings larger than the destination buffer can lead to a buffer overrun, such as the example below:

```
static char ttyenv[15], debugenv[15], *noaskenv,
           pagerenv[15], *quietenv, rootenv[25];

...

sprintf(ttyenv, "MM_NOTTTY=%d", DefinitelyNotTty);
putenv(ttyenv);
```

Here, the expanded string format is 22 bytes, and is trying to fit into a destination buffer that is 15 bytes. SmartRisk tags this as a **Severe Error**, and it should be fixed immediately.

**Format string vulnerabilities**

Variable format strings could lead to a security vulnerability where an attacker could manipulate a run-time variable and cause the format string to be replaced with something other than the intended.

The example below of a format string vulnerability is considered a **Possible Error**:

```
fprintf(stderr, msg);
```

The binary analysis discovers the trigger in this code from the format string message. The format string variable should be replaced with a static string so that a value can not be input for exploitation.

The code below is a format string vulnerability determined as an Error:

```
scanf("%s", &inbuf);
```

Because of the unbounded **%s** format string, the call to **scanf** could result in a buffer overflow. Change the code to use precision specifiers (i.e. "**%100.s**") to reduce the security risk.

**Integer overflows/underflows**

A buffer overflow example is found in our walk-through sample application, *finger*. Below are three classic buffer overflow examples.

```
int main (void)
{
    struct sockaddr_in s;
    int fd;
    char buffer[BUF];
    int buflen;
    size_t n_read;
    char host[200];
    char user[50];
    char inbuf[100];
    char msg[1024];

    printf("\nhost: ");
    scanf("%s", &inbuf);
    strcpy(host, inbuf);

    printf("\nuser: ");
    scanf("%s", &inbuf);
    strcpy(user, inbuf);
```

In the **Severe Error** code, we discover the value of the source, **inbuf**, is 100 bytes, where the destination, **user**, is only 50 bytes. The threat of an overflow is apparent. An attack could exploit this code and take over the application. To reduce the risk, the values should be the same.

The two Errors are the same, containing an unbounded **%s** format string that results in a buffer overflow. Once again, the use of precision specifiers will mitigate this problem.

Here is another example:

```
{
char *t, *newcopy;
int len;
...
t = index(s, '\n');
...
len = t ? (t-s+1) : (strlen(s)+1);
...
if (!newcopy) ExitWithError(nomem);
strncpy(newcopy, s, len);
```

In this case, the **strncpy()** function expects an unsigned integer for the third argument, however a signed integer was passed, and could lead to an integer overflow/underflow issue.

An underflow is just the opposite.

Here is another example of an integer overflow/underflow issue:

```

APR_DECLARE(void) apr_array_cat(apr_array_header_t *dst,
                               const apr_array_header_t *src)
{
    int elt_size = dst->elt_size;

    if (dst->nelts + src->nelts > dst->nalloc) {
        int new_size = (dst->nalloc <= 0) ? 1 : dst->nalloc * 2;
        char *new_data;

        while (dst->nelts + src->nelts > new_size) {
            new_size *= 2;
        }

        new_data = apr_palloc(dst->pool, elt_size * new_size);
        memcpy(new_data, dst->elts, dst->nalloc * elt_size);

        dst->elts = new_data;
        dst->nalloc = new_size;
    }

    memcpy(dst->elts + dst->nelts * elt_size, src->elts,
           elt_size * src->nelts);
    dst->nelts += src->nelts;
}

APR_DECLARE(apr_array_header_t *) apr_array_copy(apr_pool_t *p,
                                                  const apr_array_header_t *arr)
{
    apr_array_header_t *res =
        (apr_array_header_t *) apr_palloc(p, sizeof(apr_array_header_t));
    make_array_core(res, p, arr->nalloc, arr->elt_size, 0);

    memcpy(res->elts, arr->elts, arr->elt_size * arr->nelts);
    res->nelts = arr->nelts;
    memset(res->elts + res->elt_size * res->nelts, 0,
           res->elt_size * (res->nalloc - res->nelts));
    return res;
}

```

There are three Possible Errors listed in this section of code. Each one revolves around a memory copy issue where the **memcpy()** function expects an unsigned integer for the third argument, but a signed integer was passed instead. This scenario can lead to an integer overflow/underflow.

## Reviewing Results

Results of the security scans are posted in a **Summary Report** and a **Detailed Report**. Summary Reports illustrate the types of discovered graphically, showing you in bar graphs how one severity rating measures to another, and an Adjust Risk Point average for security success measurement.

The Detailed Reports are the core output for the results of the security scans. Detailed Reports describe the type of flaw the code is, its location in the application, and suggestions to fix. Detailed Reports can also be edited by managers and developers to add notes or make corrections for security corrections.

The results should be reviewed by those in the development team who are responsible for the insecure code and those who are fixing the vulnerabilities. The results aid development teams not only in fixing flaws, but pointing out common mistakes in secure coding.

For more information on reports, please see [Chapter 6](#).

## Editing Results

Discovered security flaws and their results can be edited. Notes can also be added to the SmartRisk Analyzer project to be included in an review document.

SmartRisk details the discovered vulnerabilities, however, developers may need to add comments. For example, a section of code flagged to be vulnerable in one application may exist in other applications. The developer may want to add a note to the development team that other applications should have this code fixed as well. Also, development teams may find flaws in their own code, either security or usability. You can add notes to point out where problems or inconsistencies may exist.

## Exporting Results

Once the results are posted, SmartRisk can export the results into a Reviewer document (\*.srf file). Reviewer documents can be reviewed in both Analyzer and Reviewer, as well as editing notes. The Reviewer version is installed on development systems, and do not require the same system requirements as Analyzer.

## Interpreting SmartRisk Results

SmartRisk categorizes the results by their severity level. Each level determines by how vulnerable the code is to an attack. As such, you should prioritize your security fix development by the severity level, starting with the highest.


For instance, consider a Severe Error a more serious security vulnerability than code marked as a Warning Error. The Warning Error could perhaps become exploited, and the code should be manually reviewed.

## Prioritizing Security Vulnerabilities

Development teams have narrow openings in their development schedule for maintenance, such as security and bug fixes, before the software release. SmartRisk prioritizes the security vulnerabilities by categorizing each flaw with a severity level. Severity levels are determined by how vulnerable the code is to an attack.

The discovered security flaws are listed in the Annotation List.

Severity	ID	Location	Title	Description
❌ Severe Error	2	finger.c.markup, Line 32 (id=35676)	strcpy	This call to strcpy() appears to contain a buffer o
❗ Error	7	finger.c.markup, Line 68 (id=43855)	sprintf	This call to sprintf() contains a potential buffer ov
❗ Error	5	finger.c.markup, Line 47 (id=44649)	sprintf	This call to sprintf() contains a potential buffer ov
❗ Error	4	finger.c.markup, Line 31 (id=35761)	scanf	This call to scanf() contains a potential buffer ov
❗ Error	3	finger.c.markup, Line 27 (id=35734)	scanf	This call to scanf() contains a potential buffer ov
⚠ Possible Error	6	finger.c.markup, Line 59 (id=40616)	sprintf	Be aware of format string errors when calling spr
⚠ Warning	8	finger.c.markup, Line 45 (id=39158)	gethostby...	DNS results can easily be forged by an attacker
⚠ Warning	1	finger.c.markup, Line 89 (id=46003)	fprintf	Calling fprintf() with a variable format string can l

Results  Annotation List

The simplest way to prioritize fixing security vulnerabilities is to list the discovered flaws by their severity. A Severe Error are the most vulnerable to an attack, and should be fixed immediately, followed by Error Alerts, Possible Errors, Warning Alerts, and Informational Alerts.

SmartRisk gives fix suggestions to each discovered security flaw, so the original programmers do not necessarily have to act as security fixers.

## Interactive Exploration

Take a look at the *finger* program again. Code is tagged with severity levels, however, we must look within the application to find the variable that affects the trigger.

The line below is a Severe error.

```
strcpy(user, inbuf);
```

The **strcpy** format string is a trigger, but that does not necessarily mean the code is automatically a security flaw. In fact, we can not tell just by looking at this single line. Now look at a larger portion of the code in this application, including the troublesome line.

```

int main (void)
{
    struct sockaddr_in s;
    int fd;
    char buffer[BUF];
    int buflen;
    size_t n_read;
    char host[200];
    char user[50];
    char inbuf[100];
    char msg[1024];

    printf("\nhost: ");
    scanf("%s", &inbuf);
    strcpy(host, inbuf);

    printf("\nuser: ");
    scanf("%s", &inbuf);
    strcpy(user, inbuf);

```

The **Severe Error** is the last line with the red X.

First, we want to find the values of the destination, **user**, and the source, **inbuf**. Finding these values, we immediately discover the problem. The source buffer is 100, while the destination is only 50, causing a buffer overflow. Buffer overflows are considered serious security hazards in applications. The programmer made an unwise choice to leave the code as written, and while the application functionally works, it is vulnerable to an attack.

There are two other pieces of code that are tagged as **Errors**. Both are the same **scanf** strings with unbounded **%s** strings. Because of the unbounded **%s**, the call to **scanf** contains a potential buffer overflow. Use format precision specifiers to fix both security flaws.

The series of code directly above is near identical to the section with the Severe error, however, **strcpy** in this section is not an error. Here, the source buffer, **inbuf** is still 100, but the destination, **host**, is 200, so there is no buffer overflow. That code is correct, and not a security vulnerability. The **scanf** string is still a flaw because of the unbounded **%s**.

## Saving an Analysis

Once you have successfully run a binary analysis and security scan on your project, save the results to a SmartRisk Analyzer Project file so that you can reference the application's security vulnerabilities in the future without having to run a full binary analysis again.

Under the File menu, click **Save Project** to save your created project. This saves the entire binary analysis, the security scan, and any notes you may have edited during the session.

Your SmartRisk project file (\*.saf) **must** be saved and stored in the top level directory for the application in which you are analyzing. For example, if you project has the following directory structure:

```
\ntp
\ntp\ntpd
\ntp\libntpd
```

and you are analyzing the binary file `\ntp\ntpd\ntpd`, which used a library source in `\ntp\libntpd`, you would need to put your.saf project file in `\ntp`.

SmartRisk Analyzer conducts searches downwards through the directory structure to find the required libraries and source files.

**Note:** Be aware that SmartRisk Analyzer Project files can be large. The size of the project file depends on the size of your analyzed application.

## SmartRisk Environment Files

The compiler set the environment setting depending on the type of application you are developing. SmartRisk uses that information to determine what to analyze, and how to build the appropriate models during binary analysis.

### Selecting Existing Environment Files

As SmartRisk initially opens your application to read it for compatibility, it also scans for certain pieces of information, including the environment files used to compile the application.

SmartRisk determines what parts of the application code is original (i.e. created by you or your development team) and what code is generated from the manufacturer of the compiler you used (i.e. Microsoft for Visual Studio). SmartRisk does not by default include the pre-generated code, and only loads the original code of your application for analysis.

Because SmartRisk does not include pre-generated code, the workload for the binary analysis is much less. SmartRisk Analyzer finds problems anywhere in your program. SRA can only report on source you have available.

# CHAPTER 6: REPORTING

SmartRisk Analyzer publishes security reports in two formats:

- Summary Reports
- Detailed Reports

Both reports describe the type of security flaws in their applications, where the flaws are located, and how to fix them.

## Getting a Report

Once SmartRisk Analyzer completes **Binary Analysis**, run a **Full Security Scan** on your application. Click **FullSecurityScan** in the **Global Scans** submenu under the **Scans** menu. The scan takes only a few moments.



Once the security scan completes, the **Summary Report** appears in the main window to the right. The **Detailed Report** is in the Output window at the bottom of SmartRisk Analyzer.

## Summary Reports

Summary Reports:

- Give a brief, overall picture of the discovered security vulnerabilities in a simple format.
- Presents results with bar graphs.
- Helps users find the most immediate problems with little research.

The Summary Report is the graphical presentation of the security scan. The Summary Report is located in the main window of SmartRisk Analyzer, under a tab from a copy of the original source code.

# SmartRisk Analyzer Summary

**Risk Points:** 68  
**Adjusted Risk Points:** 68  
**Lines of Code:** 90

### Results by Severity

Severe Error	1	
Error	2	
Possible Error	6	
Warning	2	
Information	0	

### Results by Type, Sorted by Risk

Format String Issues	8	
Buffer Overruns	2	
Privilege Escalation	1	

The volume of security remediation for a development team can be assessed with a quick glance at the Summary Report. The report lists the amount of **Risk Points**, generated by the number of severities and their point value. The **Adjusted Risk Points** is a value of the Risk Points divided by the lines of code in the application. The higher the Adjust Risk Points, the more security problems the application contains. The final value is the number of **Lines of Code** in the application.

The **Results by Severity** and the **Results by Type, Sorted by Risk** are at the bottom of the display, with line bars and the number of errors for each category.

### Results by Severity

SmartRisk Analyzer lists **Results by Severity**, starting with the highest, and gives the number of errors for each level. The graphical bars measure the volume of errors by that severity level in comparison to the others.

### Results by Type

SmartRisk lists the **Results by Type** starting with the risk with the most cases. The graphical bar measures the volume of the risks per severity.

Both sets of results give a measurement to the volume of security fixes a development team will face before releasing their software application to the customer.

## Detailed Reports

Detailed Reports:

- Describe type, location, and severity of the flaws.
- Direct users to the exact area in the source where the flaw is located.
- Describe why the code is a security vulnerability and suggests a fix.

The Detailed Report is the core output from the Binary Analysis and the Security Scan. View the **Annotation List** at the bottom of the screen to find the discovered errors. The list is sortable.

Severity	ID	Location	Title	Description	Creator
Severe Error	3	finger.c.markup, Line 32 (id=46117)	strcpy	This call to strcpy() contains a b...	Scripted Analysis
Possible Error	10	finger.c.markup, Line 68 (id=46231)	sprintf	This call to sprintf() contains a p...	Scripted Analysis
Possible Error	9	finger.c.markup, Line 59 (id=46207)	sprintf	Calling sprintf() with a variable fo...	Scripted Analysis
Possible Error	8	finger.c.markup, Line 47 (id=46151)	sprintf	This call to sprintf() contains a p...	Scripted Analysis
Possible Error	7	finger.c.markup, Line 72 (id=46238)	printf	Calling printf() with a variable for...	Scripted Analysis
Possible Error	6	finger.c.markup, Line 69 (id=46233)	printf	Calling printf() with a variable for...	Scripted Analysis

Detailed Reports provide comprehensive information on the discovered flaws, providing as much information as SmartRisk Analyzer could interpret in the deep binary modeling.

**Severity** is the level of the security flaw, listed from highest to least by default, and each security flaw is given an **ID** for identification, as well as the **Location** of the source file and line. The **Title** of the flaw is the format string in the code that contains a security error, with a **Description** of what is wrong, why it is a security vulnerability, and suggests a fix. The **Creator** is the user running the analysis. Creators are edited under the **Settings** menu.

**Double-click** the Severity and SmartRisk jumps to the flawed code in a copy of the original source code.

**Right-click** the severity in the Annotation List to Edit or Remove the Note.

**Click** the severity icon next to the flawed code to view the Note. The Note displays all the information in the Detailed Report.

## Exporting Results to a Generated Report

The results of the analysis can be exported to a Review document to be viewed in SmartRisk Analyzer Reviewer. Reviewer separates itself from Analyzer in that it does not run a binary analysis or security scans. Reviewer versions of SmartRisk Analyzer can be installed on development machines to view results for corrective action.

To generate a review document in Analyzer:

- Click **Generate Review Document** from the **Reader** menu in SmartRisk Analyzer.
- Name your **\*.srf** file and save it to disk.

- Once the reviewer document has been generated, you can add notes and additional information for the development team.

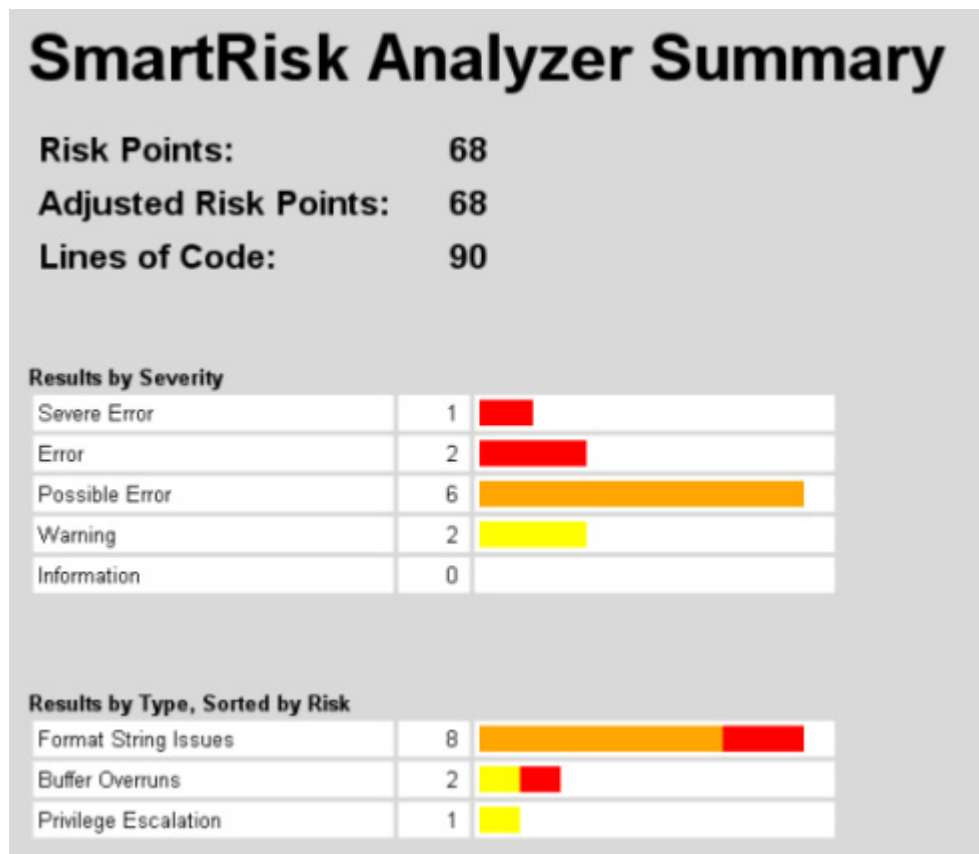
Review documents are also smaller in size than SmartRisk Project files. Review documents include copies of the source files, while the project files contain more information to run a binary analysis.

## Reading Reports

Both Analyzer and Reviewer of SmartRisk can view Summary and Detailed reports, add notes to a report, and view a copy of the original source annotated with severity levels. While both reports derive from the same binary analysis, they are presented differently to give you separate view points of the discovered vulnerabilities.

The Summary Report describes the discovered problems graphically, delivering a quick statement about the security status of your analyzed application. The **Adjusted Risk Points** gives a value that can be used to rate the analyzed application against other projects. The report breaks down the number of flaws by their severity, as well as how many by the type of security flaw. This information can be used in status reports and to schedule work throughout the development team.

For instance, look at the sample Summary Report below:



By looking at the Summary Report, we can determine a few important details immediately:

- The Adjusted Risk Points value is very high, meaning overall this application contains a high-rate of security flaws.
- This is a small program, but there are 11 discovered flaws, the majority rated as Possible Errors.
- Most of the Possible Errors found are format string issues.

If we were to use this report to schedule fixes for our application and improve code writing in our development team, we would look at the two most severe errors: the one **Severe Error** and the two **Errors**.

First the **Severe Error**. If we look in the copy of the original source by double-clicking the Severe Error listed in the **Detailed Report**, SmartRisk Analyzer labels this code as a Severe Error:

```
strcpy(user, inbuff);
```

This code is a security flaw because it contains a buffer overflow. If we reference the graph, there are also two other flaws regarded as a Buffer Overrun, but this is more serious. In comparison to the size of our application, the number of flaws in this type of risk is low, and they may also appear in other applications the development team has worked on in the past. It might be advantageous to investigate applications customers are actively using.

The second problem is the two Errors, both seem to be regarded as **Format String Issues**. Nearly three-quarters of the discovered security flaws are Format String Issues, telling us that the development team did a poor job in this area, and needs to write more secure code.

We need to look at the Detailed Report to see if these format string issues are identical, but we can assume they are similar. Having this many errors in one application could result in multiple attacks, even though they are ranked as Possible Errors. Once an attacker discovers that several of the vulnerabilities are format strings, attacking the application could be easier and more frequent.

For more information on the results, look at the Detailed Report. Below is a capture of the Annotation List for this analysis.

Severity	ID	Location	Title	Description
❌ Severe Error	8	finger.c.markup:main, Line 32 [id=22620]	strcpy	This call to strcpy() contains a buffi
❗ Error	10	finger.c.markup:main, Line 31 [id=12868]	scanf	This call to scanf() contains a pote
❗ Error	9	finger.c.markup:main, Line 27 [id=7264]	scanf	This call to scanf() contains a pote
⚠️ Possible Error	7	finger.c.markup:errorprint, Line 89 [id=39827]	fprintf	Calling fprintf() with a variable form:
⚠️ Possible Error	6	finger.c.markup:main, Line 68 [id=29026]	sprintf	This call to sprintf() contains a pote
⚠️ Possible Error	5	finger.c.markup:main, Line 59 [id=29196]	sprintf	Calling sprintf() with a variable form
⚠️ Possible Error	4	finger.c.markup:main, Line 47 [id=5506]	sprintf	This call to sprintf() contains a pote
⚠️ Possible Error	3	finger.c.markup:main, Line 72 [id=41983]	printf	Calling printf() with a variable forma
⚠️ Possible Error	2	finger.c.markup:main, Line 69 [id=41644]	printf	Calling printf() with a variable forma
⚠️ Warning	11	finger.c.markup:main, Line 45 [id=14387]	gethostbyname	This function relies on DNS entries
⚠️ Warning	1	finger.c.markup:main, Line 75 [id=18821]	recv	This use of the recv() function app

This is only a partial list, but we can see repetition in the **Titles** and **Descriptions**. Double-click the icon to jump to the code in the original source. Below is a section of code with a **Possible Error** highlighted.

```

if(!he) {
    sprintf(buffer,"can't resolve %s\n",host);
    errorprint(buffer);
    exit(1);
}
s.sin_addr=*(struct in_addr*)he->h_addr_list[0];
}
if(connect(fd,(struct sockaddr*)&s,sizeof(s))== -1){
    perror("connect");
    exit(1);
}
strcat(user,"\r\n");

sprintf(buffer,user);

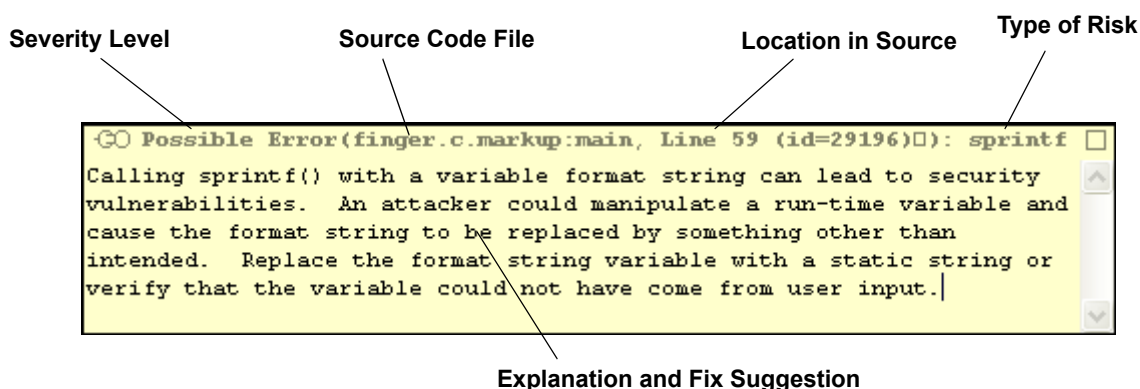
buffer[BUF-1]='\0';
buflen = strlen(buffer);
if(send(fd,buffer,buflen,0)== -1){
    perror("write");
    exit(1);
}

sprintf(msg, "fingering %s at %s\n", user, host);
printf(msg);
sprintf(msg, "IP: %d.%d.%d.%d\n", s.sin_addr.S_un.S_un_b.s_b1, s.
    s.sin_addr.S_un.S_un_b.s_b3, s.
printf(msg);

```

There are several errors in this area of the source code. Looking at the Note on each, all the flaws are the similar enough to see that the programmer was consistent in writing insecure code.

The note for the highlighted section is below:



We would have to look at each Note to see if the flaws and suggested fixes are the same, but we can assume they are similar, judging from what we already know. When scheduling security fixes in this application, we may want to assign everyone attached to the development of this project a set of format string issues to fix as a way of educating our programmers in secure coding, particularly in this area.

Notes for security flaws and other pieces of information can be added or modified by a development manager or programmer to communicate ideas to other team members.

The Summary Report and the Detailed Report gives us a clear picture on the security of our application. Between the two, we found a problem in the way we code our applications from the large number of errors and types of errors.



# CHAPTER 7: UNDERSTANDING ERROR MESSAGES






In this chapter, we take a look at the types of severity alerts SmartRisk Analyzer reports and the code associated with them.

Once SmartRisk Analyzer reports security vulnerabilities and categorizes the severity type, taking appropriate action to remedy these flaws is the goal and purpose of secure code analysis. In order to fix errors, it is important to understand the error messages SmartRisk Analyzer produces to improve your current application's security and for future development.

## Types of Severity

As mentioned in previous chapters, SmartRisk Analyzer categorizes security flaw severity as follows:

**Table 1: Five Levels of Severity**

Severity Warning	Description
<b>Severe Error</b>  Severe Error	SmartRisk Analyzer determines the code has serious security vulnerabilities and is an easy target for an attacker. You should modify the code immediately.
<b>Error</b>  Error	The code has security vulnerabilities, and should be modified immediately.
<b>Possible Error</b>  Possible Error	The code could be a security vulnerability, and become a target for an attacker. Review this code manually to determine if it contains a security vulnerability.
<b>Warning Error</b>  Warning	The code might eventually result in a security vulnerability, but does not represent a high immediate risk.
<b>Informational Alert</b>  Informational	Cross-references and informative messages.

**Note:** Only applications with zero security errors are considered flawless. All flaws should be investigated and fixed before the application is released or implemented.

## Determining Severity

SmartRisk Analyzer determines the security flaw severities as it scans your application. After the analysis completes, SmartRisk has a model of your application to reference code against a database of known security flaws. The more dangerous the flaw, the higher the severity.

SmartRisk searches the application for certain triggers during Binary Analysis. Triggers are format strings that are known to be security vulnerabilities, such as the `printf` family of code in C programming. Once the triggers are discovered, SmartRisk uses the data and control flow graphs to determine the severity of the error.

## Prioritizing Severity

SmartRisk Analyzer, by default, lists discovered security flaws by severity, starting with the highest. As such, correcting higher severities first should take precedence over lower level severities. One obvious reason is that flaws more severe are extremely vulnerable to attacks, and should be fixed immediately. Also, fixing severe errors may at times fix lower level problems, because the two are linked.

For example, the code below is an **Error**:

```
liststart = (db_entry *)calloc(1, sizeof(db_entry));
cur_entry = liststart;
```

The next piece of code is an **Informational Alert**:

```
cur_entry->next = NULL;
```

SmartRisk Analyzer notes that the Informational is a cross-reference to the code tagged as an Error. SmartRisk suggests to check the results of the `calloc()` call function before using in both instances, to prevent application instability or crashing due to unavailable memory. Once the Error is fixed, the Informational will most likely be considered secure code.

Not all flaws are related to another, however, many will be in your application. Examine each flaw in the application before fixing to learn more information about SmartRisk's discoveries.

## Severity Descriptions

SmartRisk segregates flaws by severity. Severity is determined by how dangerous the code is in the application. Severity levels are as followed, from highest to least:

- Severe Error
- Error

- Possible Error
- Warning
- Informational

SmartRisk analyzes the data flow and the control flow to measure the severity of each discovered flaw.

## Severe Errors

Severe errors identify code as a serious security flaw, and a high risk for an attack.

The code below is from the sample application *finger* that was discussed in the project walk-through in [Chapter 4](#), and is considered a Severe error:

```
strcpy(user, inbuf);
```

The severity is not determined from one line, but from where these values are directed or called from. The trigger is **strcpy**. We need to look further into the application to find out if this line is a security problem, and we start by finding the values of the source and destination buffers, **user** and **inbuf**, respectively.

```
char user[50];
char inbuf[100];
```

The value of the source buffer, **inbuf**, is 100, while the value of the destination buffer, **user**, is 50. Automatically, this code should be flagged. The programmer is allowing a larger buffer to go into the destination, causing a buffer overflow. Attackers use buffer overflows like this piece of code to take control of an application.

## Error

Code tagged as an **Error** consists of security vulnerabilities that are serious, but may be more difficult to attack. Attack on these lines of code are a realistic threat, and the code should be fixed immediately.

The code below is considered an Error.

```
scanf("%s", inbuf);
```

Here, there is a potential buffer overflow.

Using format precision specifiers tightens security. Error Alerts point out serious security flaws, such as an unbounded **%s**, which should be fixed immediately.

Here is another example of an Error level risk, this time with an unbounded **%s**:

```
sprintf(msg, "fingering %s at %s\n", user, host);
```

The **sprintf** string is a trigger, so SmartRisk looks for these types of coding automatically, moreover, it contains more unbounded **%s** values. The unbounded **%s** should have precision modifiers.

Another Error is in the following example:

```
liststart = (db_entry *)calloc(1, sizeof(db_entry));
cur_entry = liststart;
}
cur_entry->next = NULL;
```

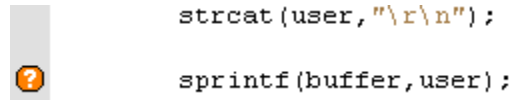
The results of the **calloc()** call are not being checked for success, which can result in application instability or crashing if memory is not available. Not all

`calloc()` strings are security vulnerabilities, however, this example does not check the results.

### Possible Error

**Possible Errors** indicate that the code has security vulnerabilities, and that attackers could intrude into the application. These errors are serious enough for a development team to dedicate time fixing. And while one may be tagged more severe, does not mean others should be ignored.

Take a look at the example below with the Possible Error already tagged.



```
strcat(user, "\r\n");
sprintf(buffer, user);
```

We cannot determine the tagged code is wrong, until we look at the value of the `buffer` and `user` directly above.

A variable format string is used, which not only can lead to security vulnerabilities, but this is how SmartRisk determined that the code needed to be flagged. In this type of example, it is worth looking at the code closer to determine whether the variable actually comes from user input, making it a serious security risk. Ideally, using a static string secures the code.

Ironically enough, the code above our Severe example is a Possible Error.

```
scanf("%s", &inbuf);
```

SmartRisk Analyzer flagged this line because of the unbounded `%s` in the format string. Secure code would attach a value to the `%s`, such as:

```
%100.s
```

Format strings like `sprintf()`, `scanf()`, and `strcpy()` appear often as insecure code because of unbounded variables.

### Warning Error

The **Warning Error** example below also contains a `printf` statement, this time an `fprintf`:

```
fprintf(stderr, msg);
```

SmartRisk flagged this code due to the variable format string, `stderr`. Once again, using a static format string mitigates the security problem. Leaving the code as written could result in access from an intruder.

Here is another example of a Warning Error.

```
he=gethostbyname(host);
```

The function relies on DNS entries, which the results can be forged by an attacker. The results can be arbitrarily set to large values, and conducting a sanity check on the results to tighten security.

### Informational Alerts

**Informational Alerts** do not necessarily indicate a security vulnerability, and some structs tagged as Informational are fixed when more serious flaws are corrected. Informational Alerts help clarify poor and weak coding, as well as cross-reference and strengthen code security.

The example below does not check the results for success, and can result in application instability.

```
cur_entry->next = NULL;
```

It is directly linked to an Error in the code above it:

```
liststart = (db_entry *)calloc(1, sizeof(db_entry));
```

The programmer should fix the Error line, and if the security results were properly prioritized, it is already fixed, which resolves the Informational Alert.

Informational Alerts instruct potential problems with the written code. The following is an example of an Informational Alert:

```
if ((pid = fork()) < 0) {
```

The trigger in this line is **fork**, and because it creates a copy of the current process's memory space, should an attacker take advantage of this weakness, the intrusion could be damaging if the application contains sensitive data.


## Multiple Errors in Code

Many security flaws are connected. If a flaw is fixed in a section of code, that fix could perhaps fix another security flaw with a lower severity level. Take the following example:

```
if (memsuite) {
    XML_Memory_Handling_Suite *mtemp;
    parser = memsuite->malloc_fcn(sizeof(Parser));
    mtemp = &((Parser *) parser)->m_mem;
    mtemp->malloc_fcn = memsuite->malloc_fcn;
    mtemp->realloc_fcn = memsuite->realloc_fcn;
    mtemp->free_fcn = memsuite->free_fcn;
}
else {
    XML_Memory_Handling_Suite *mtemp;
    parser = malloc(sizeof(Parser));
    mtemp = &((Parser *) parser)->m_mem;
    mtemp->malloc_fcn = malloc;
    mtemp->realloc_fcn = realloc;
    mtemp->free_fcn = free;
}
```

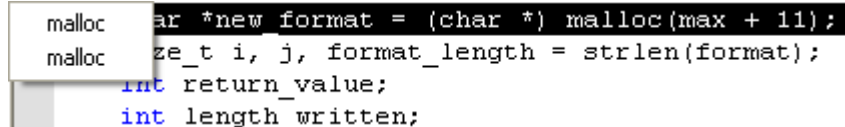
The code marked as an **Error** is a security flaw due to the memory allocation not checked for success. The **Informational** below the **Error** cross-references the memory allocation error in the code. The result could crash the application, or render instability if memory is not available. Once the **Error** is fixed, the **Informational** should be resolved as well.

Some code may contain more than one error in the same string, that is two or more variables in the same line could result in different security vulnerabilities. SmartRisk labels both with the highest severity of the two. Below is an example of an **Error** and a **Possible Error** in the same line:



```
char *new_format = (char *) malloc(max + 1);
size_t i, j, format_length = strlen(format);
int return_value;
int length_written;
```

The small table in the icon indicates that there is more than one error in this section. If we click on the icon to view the note, SmartRisk shows us both error types to choose:

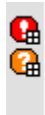


```
malloc char *new_format = (char *) malloc(max + 1);
malloc size_t i, j, format_length = strlen(format);
int return_value;
int length_written;
```

In this example, both errors are in the memory allocation format string. The **Error** in the code deals with results not being checked properly, which can lead to application instability and crashing if memory is not available.

The **Possible Error** part is because the `malloc()` function expects an unsigned integer for the first argument, but a signed integer was passed in. This leads to possible integer overflow/underflow issues.

Here is a third example that combines multiple errors the same section and the same type:



```
start_argv = malloc((argc + 1) * sizeof(const char **));
memcpy(start_argv, argv, argc * sizeof(const char **));
start_argv[argc] = NULL;
```

This section contains four flaws. The first labeled as an **Error** is an **Error** and **Possible Error**, both in the `malloc()` function. The second labeled as a **Possible Error** is a **Possible Error** and **Informational**. The **Possible Error** references the `memcpy()` function, and the **Informational** is a reference to the `malloc()` from the previous flaw.

When fixing these security flaws, we should look at all four notes. Starting with the `malloc()` function, the **Informational** note details that the results of the call is used without being checked for success. Similarly, the **Error** note details problems with checking, so we know those two flaws are directly connected. The other note we need to look at is `malloc()` labeled as a **Possible Error**, which details problems with an unsigned integer for the first argument, but a signed integer was passed in, resulting in an integer overflow/underflow issue.

When fixing the `malloc()` function, keep in mind both to check for success and that the integer can not result in an overflow by an attacker. The Error should be resolved once these are issues are abated.

The `memcpy()` function is another overflow issue. It needs to either expect the signed integer to pass, or directed to the third argument to be fixed.

When fixing a flawed string with multiple errors inside or two that are related, keep in mind both results. If the flaws are related, reference the less flaw for more information on fixing the security vulnerability. If the flaw is in the same line, or the same string, referencing both notes can help rewrite the code with stronger security.



# CHAPTER 8: SMARTRISK ANALYZER REFERENCE

This chapter discusses scan types and common triggers as security risks SmartRisk Analyzer uses to detect flaws.

SmartRisk Analyzer uses the binary analysis to look at the entire program, how it works, and in the process it finds where the application is vulnerable to a security attack. The binary analysis builds a binary model of the application by reading the data flow, the control flow, and the range propagation of the application. For more information on the binary modeling, refer to [page 2-7](#).

SmartRisk finds triggers during the long binary analysis - key strings that are known to create vulnerabilities, tracing the code to determine if it is an error, and scanning the application for types of security risks.

Part of secure coding is realizing code that could potentially be regarded as insecure code, and writing it securely before software release. This chapter covers the areas of security problems, how applications are scanned to determine security problems, and some suggestions to fixing the problems.

## Scan Types

SmartRisk Analyzer scans for the following security risks and conditions:

- Stack/Heap Buffer Overruns
- Format string vulnerabilities
- Error return checking
- Integer overflows/underflows
- Threading/race conditions
- Cryptography
- Database
- Denial of Service
- Reliability Issues
- Input Validation Issues
- Privilege Escalation
- Network Issues

SmartRisk Analyzer reports many of these conditions as security issues in analyzed applications. The most common are detailed below.

## Stack/Heap Buffer Overruns

**Description:** Buffer overruns occur when a value is copied into smaller memory space. Checks are not performed as the source is copied into the destination.

**Risk Characteristics:** Attackers can overflow the buffer with something that is unintended, and render the application unstable or crash the system, or the unintended code could be for malicious intent.

**Types of Triggers:** strcpy(), printf functions, scanf, recv

**Example:**

```
int main(int argv, char **argc) {
    extern system, puts;
    void (*fn) (char*)=(void(*) (char*)) &system;
    char buf[256];
    fn=(void(*) (char*)) &puts;
    strcpy(buf, argc[1]);
    fn(argc[2]);
    exit(1); }
```

Here, the destination, **buf**, is 256 bytes, but the source, **argc[1]**, is valued at 512 bytes. As a security vulnerability, an intruder could control the microprocessor. The trigger here is **strcpy()**. The destination buffer should be sized large enough to hold the source value, or the destination buffer larger than the source. The source cannot be larger than the destination, otherwise we receive a buffer overflow like the above. The example is provided by <http://community.corest.com/~gera/InsecureProgramming/>.

## Format String Vulnerabilities

**Description:** Variable format strings leading to security vulnerabilities, which should be replaced with a static string.

**Risk Characteristics:** Attackers could manipulate a run-time variable and replace it with something unintended, leading to exploitation of the application.

**Types of Triggers:** printf() functions

### Example:

```
char count[10];
remains = ((start + 30) - time(NULL));
sprintf (count, "%d...", remains);
if (!SetConsoleCursorPosition(hConErr, coninfo.dwCursorPosition))
return;
if (!WriteConsole(hConErr, count, strlen(count), &result, NULL)
    || !result)
    return;
}
```

The code in bold is where the security flaw starts. SmartRisk Analyzer labels this as a **Possible Error**. The **sprintf** has a variable format string, **%d**, which can lead to security vulnerabilities, should an attacker manipulate a run-time variable and cause the format string to be replaced by something other than what is intended. The code should be written with a static format string to mitigate the security problem. One solution to the security flaw could be the following:

```
char count[10];
remains = ((start + 30) - time(NULL));
sprintf (count, "%.6d...", remains);
if (!SetConsoleCursorPosition(hConErr, coninfo.dwCursorPosition))
return;
if (!WriteConsole(hConErr, count, strlen(count), &result, NULL)
    || !result)
    return;
}
```

## Error Return Checking

**Description:** Check the return value of every function that may return an error code.

**Risk Characteristics:** The program can crash due to invalidated values (reliability). There is a risk of privilege escalation if the code does not check the result of authentication functions for returned errors.

**Types of Triggers:** scanf(), recv(),

**Example:**

```
sendAuthRequest (port, AUTH_REQ_PASSWORD);  
status = recv_and_check_password_packet (port);  
sendInitialPacket (port);
```

There is some authentication going on, however, the status is not checked for error. The code should be fixed as below:

```
sendAuthRequest (port, AUTH_REQ_PASSWORD);  
status = recv_and_check_password_packet (port);  
if (status != 0)  
    graceful_exit (status, "Authentication failed." );  
sendInitialPacket (port);
```

Even ordinary, non-authentication code should be checked for errors. A common problem is not checking for a low-memory situation where a request for memory is denied. Also see **Memory Functions** on [page 8-19](#).

Any function call that could return an error should be checked for error before relying on the output of that function. Also see **recv** on [page 8-25](#) for an example proper error checking for socket input errors.

## Integer Overflows/Underflows

**Description:** Creating a buffer size larger than the data input.

**Risk Characteristics:** This can cause application instability or a system crash, or the code can be exploited and filler the buffer with large code or application.

**Types of Triggers:** malloc(), memcpy(),

### Example:

```

        if (node->val_len - offset < len) {
ptr = (char *)realloc(node->value, node->val_len + BUFSIZE);
        if (!ptr) {
            log(LOG_ERR, "Call to realloc failed");
            printf("Limits exeeded\nClosing\n");
            free(inpbuf);
            return 0;
        }
        node->value = ptr;
        node->val_len += BUFSIZE;
    }

```

The code in bold is considered a **Possible Error**. Like most **Possible Errors**, this is a potential security problem, however, due to the complexity of the application, SmartRisk cannot determine if the code is a risk. Manual analysis is necessary for risk determination. The **realloc()** function expects an unsigned integer as its second argument, however, a signed integer was passed in. The result can lead to an integer overflow/underflow issue.

```

int win32_strftime_extra(char *s, int max, const char *format,
                        const struct tm *tm)
{
    /* If the new format string is bigger than max, the result string won't fit
     * anyway. If format strings are added, made sure the padding below is
     * enough */
    char *new_format = (char *) malloc(max + 11);
    size_t i, j, format_length = strlen(format);
    int return_value;
    int length_written;

    for (i = 0, j = 0; (i < format_length && j < max);) {
        if (format[i] != '%') {
            new_format[j++] = format[i++];
            continue;

```

There are a couple of problems with this code. The **malloc()** call is not checked for success, leading to application instability or a system crash. We

forgot to check the length of the buffer size, which has caused this security vulnerability.

The `malloc()` call also expects an unsigned integer passed on the first argument, and a signed integer is passed in. If an attacker were able to exploit this code, they could create a buffer size less than the data placed inside the buffer.

## Threading/Race Conditions

**Description:** One or more processors attempting to access the application. One processor belongs to the local or network system attached to the application. The other is an intruder.

**Risk Characteristics:** Data is not locked up properly, and becomes corrupted.

**Types of Triggers:**

**Example:**

```
#include
#include

#define N 4

DWORD WINAPI hello (LPVOID myID)
{
    printf ("Hello from thread %d\n", *(int *)myID);
}

int main (int argc, char* argv[])
{
    int j;
    HANDLE h[N];
    DWORD rc;

    for (j = 0; j < N; j++)
    {
        h[j] = CreateThread (0, 0, hello, (LPVOID)&j, 0, NULL);
    }

    rc = WaitForMultipleObjects (N, h, TRUE, INFINITE);
}
```

The source of the race condition is the `j`, where it is incorrectly passed to the threaded function.

```
int id[N];
for (j = 0; j < N; j++)
{
    id[j] = j;
    pthread_create (&tid[j], NULL, hello, (void *)&id[j]);
}
```

# Cryptography

**Description:** Random generators do not use enough entropy when used for security purposes.

**Risk Characteristics:** Input with variable format strings can lead to security violations.

**Types of Triggers:**

**Example:**

```
if (randseed==0) {  
    randseed = (int)apr_time_now();  
    seedrandom(randseed);  
}
```

The example above does not use a standard random number generator. If this were used where there are security concerns, the use of a cryptographic random number generator, such as the following:

## Database

**Description:** Application reads in user-supplied data, such as a name or address, then passes it into a database. An Attacker can provide name or address that contains some SQL commands embedded in it, and maliciously manipulate the database.

**Risk Characteristics:** Check all user input for SQL commands before passing them to the database. Need more info from this from a more experienced SRA type.

**Types of Triggers:**

**Example:**

# Denial of Service

**Description:** Vulnerable code that enables an attacker to send commands of unlimited length into an application and run it out of memory, rendering the program from functioning.

**Risk Characteristics:** Denial of Service vulnerabilities can allow an attacker to send commands of unlimited length to stop the program from performing its function.

## Types of Triggers:

### Example:

```

/*
 * qmail-dos-1 - run a qmail system out of swap space by feeding long SMTP
 * commands.
 *
 * Usage: qmail-dos-1 hostname
 *
 * Author: Wietse Venema. The author is not responsible for abuse of this
 * program. Use at your own risk. Batteries not included.
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdarg.h>
#include <errno.h>
#include <stdio.h>

void    fatal(char *fmt,...)
{
    va_list ap;

    va_start(ap, fmt);
    vfprintf(stderr, fmt, ap);
    va_end(ap);
    putc('\n', stderr);
    exit(1);
}

int     main(int argc, char **argv)
{
    struct sockaddr_in sin;
    struct hostent *hp;
    char    buf[BUFSIZ];
    int     sock;
    FILE    *fp;

```

```

if (argc != 2)
    fatal("usage: %s host", argv[0]);
if ((hp = gethostbyname(argv[1])) == 0)
    fatal("host %s not found", argv[1]);
memset((char *) &sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
memcpy((char *) &sin.sin_addr, hp->h_addr, sizeof(sin.sin_addr));
sin.sin_port = htons(25);
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    fatal("socket: %s", strerror(errno));
if (connect(sock, (struct sockaddr *) &sin, sizeof(sin)) < 0)
    fatal("connect to %s: %s", argv[1], strerror(errno));
if ((fp = fdopen(sock, "r+")) == 0)
    fatal("fdopen: %s", strerror(errno));
if (fgets(buf, sizeof(buf), fp) == 0)
    fatal("connection lost");
memset(buf, 'X', sizeof(buf));
fseek(fp, 0L, SEEK_SET);
while (fputs(buf, fp) != EOF)
    /* void */ ;
}

```

This example shows a **Denial of Service** error that could be exploited by an attacker. Here, the intruder could send the application SMTP commands of unlimited length to run the application of memory and crash the machine. The recommendation to this type of vulnerable code is to add bounds on the amount of data that is read per command.

# Reliability Issues

**Description:** Insecure code where an attack could result in an unstable application or system crash if memory becomes unavailable.

**Risk Characteristics:** Memory buffers not checked could become vulnerable, and filled with unlimited buffer sizes.

**Types of Triggers:** malloc()

**Example:**

```
db = malloc(sizeof(*db));  
memset(db, 0, sizeof(*db));
```

In the example, above the bold line is the point SmartRisk determines an security flaw starts, and considers the risk severity an **Error**. The **malloc()** call is not checked for success before being used. To mitigate this problem, modify the results so that the code looks as follows:

```
db = malloc(sizeof(*db));  
If (db == NULL)  
    error_exit("Memory allocation error");  
memset(db, 0, sizeof(*db));
```

Here is another example:

```
start_argv = malloc((argc + 1) * sizeof(const char **));  
memcpy(start_argv, argv, argc * sizeof(const char **));  
start_argv[argc] = NULL;
```

This example contains many vulnerabilities, most of all a **Reliability Issue** with the **malloc()** function. Like most problems with the **malloc()** function, this one is not checked for results, which can result in application instability or system crash. The **malloc()** function also contains an integer overflow/underflow issue, because it expects an unsigned integer for the first argument, but a signed integer is passed in.

```
Size_t alloc_size = (argc + 1) * sizeof(const char **);  
if ((start_argv = malloc(alloc_size)) == 0)  
    error_exit("Memory allocation error");  
size_t arg_size = argc * sizeof(const char **);
```

```
// we can see arg_size is less than alloc_size right above, but if the source  
// could be larger than the destination, limit the number of bytes copied.
```

```
memcpy(start_argv, argv, arg_size);  
start_argv[argc] = NULL;
```

## Input Validation

**Description:** Code with environment variables that contain untrusted input. The environment string is pointed to by *name*, and returns the associated value to the string, but this returned value should not be written to.

**Risk Characteristics:** Attackers use this code to cause an application to behave in an unintended manner.

**Types of Triggers:** `getenv()`

**Example:**

```
if (env_temp = getenv("SystemRoot")) {  
    apr_table_addn(e, "SystemRoot", env_temp);  
}
```

This example contains environment variables. These variables contain untrusted input that can be easily manipulated. An attacker could use this to cause an application to behave in an unintended manner.

# Privilege Escalation

**Description:** Unsafe code allowing access to files without necessary permissions.

**Risk Characteristics:** Allow access to control settings and files with too many privileges.

**Types of Triggers:** umask()

**Example:**

```
he=gethostbyname (host) ;
```

```
old_mode = umask(077) ;  
*fd = mkstemp(pattern) ;  
umask(old_mode) ;
```

The two lines in bold are both **Possible Errors**. The **umask ()** function can create files with unsafe privileges. Because these allow access and control settings, they should be checked to make sure that files with minimal privileges are required to accomplish the task.

## Network Issues

**Description:**

**Risk Characteristics:**

**Types of Triggers:**

**Example:**


## Common Triggers as Security Risks

Not all triggers are security flaws. If code is written securely, regardless of the format strings and structures used, the code is secure. However, there are format strings that are commonly written insecurely, and SmartRisk Analyzer scans for those strings to determine the security risk.


The trigger alone does not create a security flaw, but the code and variables associated with the trigger contribute to the security vulnerability.

Triggers are referenced as **Title** in the Detail Reports under the Annotation List.

Trigger

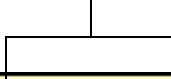



Severity	ID	Location	Title	Description
Severe Error	8	finger.c.markup:main, Line 32 (id=22620)	strcpy	This call to strcpy() contains a buffer overflow
Error	10	finger.c.markup:main, Line 31 (id=12868)	scanf	This call to scanf() contains a potential format string
Error	9	finger.c.markup:main, Line 27 (id=7264)	scanf	This call to scanf() contains a potential format string
Possible Error	7	finger.c.markup:errorprint, Line 89 (id=39827)	fprintf	Calling fprintf() with a variable format string
Possible Error	6	finger.c.markup:main, Line 68 (id=29026)	sprintf	This call to sprintf() contains a potential format string
Possible Error	5	finger.c.markup:main, Line 59 (id=29196)	sprintf	Calling sprintf() with a variable format string
Possible Error	4	finger.c.markup:main, Line 47 (id=5506)	sprintf	This call to sprintf() contains a potential format string
Possible Error	3	finger.c.markup:main, Line 72 (id=41983)	printf	Calling printf() with a variable format string
Possible Error	2	finger.c.markup:main, Line 69 (id=41644)	printf	Calling printf() with a variable format string
Warning	11	finger.c.markup:main, Line 45 (id=14387)	gethostbyname	This function relies on DNS entries
Warning	1	finger.c.markup:main, Line 75 (id=18821)	recv	This use of the recv() function appears to be a typo

Results  Annotation List

Triggers are also listed in the sticky notes for each error.

Trigger



 Possible Error(finger.c.markup:main, Line 59 (id=29196)): sprintf

Calling sprintf() with a variable format string can lead to security vulnerabilities. An attacker could manipulate a run-time variable and cause the format string to be replaced by something other than intended. Replace the format string variable with a static string or verify that the variable could not have come from user input.

Below are triggers in code that are known to be problematic.

# gets

**Language:** C/C++

**Platform:** All Platforms

**Description:** Reads a line from **stdin** into the buffer pointed to by the input parameter, until either a terminating newline or **EOF**, which it replaces with `'\0'`. No check for buffer overrun is performed.

**Variants:** gets, getopt, getenv, gethostbyname, fgets

**Known Types of Problems:** Input Validation Issues

**Example:**

```
char AnsBuf[100], *s;
...
s = gets(AnsBuf);
```

The **Error** in the **gets()** function receives unchecked user input, because this function does not check the size of the buffer. Use the **fgets()** function instead, since that will check the size of the buffer, and mitigate a possible buffer overflow. The **fgets()** function gets a string from a file.

```
if( fgets( AnsBuf, 100, stdin ) == NULL)
    // Handle error or bend of file
else
    printf( "%s.100", AnsBuf);
```

# Memory Functions

<b>Language:</b>	C/C++
<b>Platform:</b>	All
<b>Description:</b>	Function to allocate memory.
<b>Variants:</b>	alloc, malloc, calloc, realloc
<b>Known Types of Problems:</b>	Reliability Issues, Buffer Overflow/Underflow Issues, Privilege Escalation Issues,
<b>Example:</b>	

Below is an example of a **malloc()** call used without being checked for success.

```

sbuf = (TlsGetValue)(tlsid);
if (!fh || !sbuf) {
    sbuf = (malloc)(1024);
    (TlsSetValue)(tlsid, sbuf);
    sbuf[1023] = '\0';
}

```

SmartRisk Analyzer determines the code in bold as an **Error**, a **malloc()** call being used without being checked for success. The correct amount of memory is not properly allocated, and the result is an application crash, because the return to the pointer was NULL. The code should look as follow:

```

sbuf = (TlsGetValue)(tlsid);
if (!fh || !sbuf) {
    sbuf = (malloc)(1024);
    (TlsSetValue)(tlsid, sbuf);
    sbuf[1024] = '\0';
}

```

Now 1024 bytes of memory is properly allocated to the application.

Here is another examples:

```

int envc;
for (envc = 0; _environ[envc]; ++envc) {
    ;
}
env = malloc((envc + 2) * sizeof(char*));
memcpy(env, _environ, envc * sizeof(char*));
apr_snprintf(pidbuf, sizeof(pidbuf), "AP_PARENT_PID=%i", parent_pid);
env[envc] = pidbuf;
env[envc + 1] = NULL;

```

The results in this **Error** are not checks for success in the **malloc()** call. The value is 1 byte different, which can lead to application instability and unavailable memory. Since the result is NULL, the result fails. The value is **env[envc + n]**, the value is **n** should be the same for both statements.

```
int envc;  
for (envc = 0; _environ[envc]; ++envc) {  
    ;  
}  
env = malloc((envc + 2) * sizeof (char*));  
memcpy(env, _environ, envc * sizeof (char*));  
apr_snprintf(pidbuf, sizeof(pidbuf), "AP_PARENT_PID=%i", parent_pid);  
env[envc] = pidbuf;  
env[envc + 2] = NULL;
```

## memcpy

**Language:** C/C++

**Platform:** All

**Description:** Copies memory into a buffer.

**Variants:**

**Known Types of Problems:** Integer Overflow/Underflow Issues

**Example:**

```
if (APR_SUCCESS == apr_filepath_get(&cwd, APR_FILEPATH_NATIVE, p)) {
    if (test_tempdir(cwd, p)) {
        memcpy(global_temp_dir, cwd, strlen(cwd) + 1);
        goto end;
    }
}
```

The **Possible Error** is in bold. The **memcpy ()** function expects an unsigned integer passed for the third argument. A signed integer is passed instead, leading to a possible integer overflow/underflow issue. To mitigate this security vulnerability, the unsigned integer should be passed for the third argument as expected. The code should look as follows:

```
if (APR_SUCCESS == apr_filepath_get(&cwd, APR_FILEPATH_NATIVE, p)) {
    if (test_tempdir(cwd, p)) {
        memcpy(global_temp_dir, cwd, (size_t)(strlen(cwd)+1));
        goto end;
    }
}
```

# printf

**Language:** C/C++

**Platform:** All Platforms

**Description:** Sends formatted output.

**Variants:** vprintf, sprintf, fprintf, vsprintf, vsnprintf, vfprintf

**Known Types of Problems:** Buffer Overflows/Underflows, Format String Issues.

A **printf** statement prints out input to the user. Commonly, **printf** statements contain unbounded variables within the string. Unbounded format strings can lead to buffer overflows and underflows. When writing secure code, use format precision specifiers instead of an unbounded variable, such as a **%s**.

Here is an example of a **sprintf** statement containing an unbounded **%s**:

```
sprintf(query_env, "QUERY_STRING=%s", query_string);
```

To write this as secure code, use a precision modifier, such as **%.100s** to give the string a maximum amount of characters.

```
sprintf(query_env, "QUERY_STRING=%.100s", query_string);
```

Another type of security risk in **printf** functions is variable format string issues. If the application is violated, and the format string variable comes from user input, an attacker could manipulate a run-time variable and replace the format string with something other than intended. In the example below, the format string variable should be replaced with a static string, or verify that the variable could not come from user input.

```
sprintf (count, "%d...", remains);
```

Replacing the variable with a static string secures the code, such as the following:

```
sprintf (count, "%.6d...", remains);
```

If the variable comes from user input, attackers will not be able to manipulate the code for exploitation. Coding with variable format strings are common in applications, because variables are convenient and quick. Programmers may use a variable rather than a static string, simply because the value has not been determined when initially writing the program, or they know the program will evolve and change over a series of releases. The variable string reduces the unit testing and any chances for application inoperability, but it leaves serious security flaws that could be dangerous to customers when they implement the application on their network.

The code below is a **Severe Error**:

```
sprintf(msg, "IP: %d.%d.%d.%d\n", s.sin_addr.S_un.S_un_b.s_b1, s.sin_addr.S_un.S_un_b.s_b2,
        s.sin_addr.S_un.S_un_b.s_b3, s.sin_addr.S_un.S_un_b.s_b4);
```

The expanded format string is 53 bytes, and the destination is 20 bytes, leading to a buffer overflow. The source should be the same size as the destination, or the

destination can be larger, however, if the source is larger than a destination, it will contain a buffer overflow.

## rand

**Language:** C/C++

**Platform:** All

**Description:** Used for random number generators, authentication, and cryptography.

**Variants:** rand, srand,

**Known Types of Problems:** Cryptography, Privilege Escalation

**Example:**

```
void neo_seed_rand(int seed)
{
    srand48 (seed) ;
    RandomInit=1;

    return;
}
```

In the example above, the standard random generators do not provide a sufficient amount of entropy when used for security purposes. If this is used where security is a concern, use a trusted cryptographic random number generator instead.

## recv

**Language:** C/C++

**Platform:** All

**Description:** Receives input from user.

**Variants:** recv, recvform

**Known Types of Problems:** Buffer overruns

**Example:**

```
while ( (n_read=recv(fd,buffer,BUF,0))>0)
    fwrite(buffer,n_read,1,stdout);
if(n_read== -1){
    perror("read");
    exit(1);
```

In this example, **recv()** appears to be used in a loop, and if the buffer boundaries are not check properly, could result in a buffer overrun. To avoid security problems, check that the boundaries are limited.

## scanf

**Language:** C/C++

**Platform:** All Platforms

**Description:** Reads formatted input.

**Variants:** fscanf, sscanf, vscanf, vsscanf, vfscanf

**Known Types of Problems:** Format String Issues, Buffer Overflows

**Example:**

```
scanf("%s", &inbuf);
```

The example is an **Error**, and contains a **Buffer Overflow**. The `%s` is unbounded, and should an attacker exploit this code, they could insert large amounts of code to crash the system or take control. Using `%s` is dangerous, and programmers should replace these with precision modifiers, such as `%.100s` to secure the code and eliminate the risk of a buffer overflow. the precision modifier should also match the size of the buffer.

```
scanf("%.100s", &inbuf);
```

# strcpy

**Language:** C/C++

**Platform:** All Platforms

**Description:** Copies any specified string into a buffer.

**Variants:** strcpy

**Known Types of Problems:** Buffer Overflow/Underflow

**Example:**

```
int main (void)
{
    ...
    char host[200];
    char user[50];
    char inbuf[100];
    char msg[1024];
    ...
    strcpy(user, inbuf);
```

The code in bold is tagged as a **Severe Error** by SmartRisk Analyzer, because it contains a buffer overflow. The source buffer is 100 bytes, where as the destination buffer is 50 bytes. To fix this problem, both buffers should be 100 bytes in length. Also, **strcpy** should be changed to **strncpy**, because **strncpy** limits the number of characters copied. The destination buffer can be larger than the source.

```
int main (void)
{
    ...
    char host[200];
    char user[100];
    char inbuf[100];
    char msg[1024];
    ...
    strncpy(user, inbuf);
```

This is now secure code, because the destination buffer accommodates the source buffer without causing an overflow issue. When programming in this type of case, make certain the destination buffer (*user* in this example) is at least as large as the input buffer.

## syslog

**Language:** C/C++

**Platform:** Solaris

**Description:** Writes system log information.

**Variants:** syslog

**Known Types of Problems:** Format String Issues

**Example:**

```

syslog(level, syslog_mem);
...
switch (level) {
case LOG_ERR:
    fprintf(stderr, "[ERROR]: ");
    break;
case LOG_WARNING:
    fprintf(stderr, "[WARNING]: ");
    break;
case LOG_INFO:
    fprintf(stderr, "[INFO]: ");
    break;
case LOG_DEBUG:
    fprintf(stderr, "[DEBUG]: ");
    break;
}

```

The code in bold is an **Error**, a variable format string issue that could lead to security vulnerabilities where an attacker could manipulate a run-time variable and cause the format string to be replaced with something other than intended, such as buffers of unlimited length or false information to a log file for manipulation. The format string variable should be replaced with a static string.

```

syslog(LOG_CRIT, syslog_error_template_with_user_input, ldap_host,
unchecked_caller_input);

```

The problems with **syslog** are very similar to the problems with **printf**. The fix is similar too:

```

syslog(LOG_CRIT, "Failed to connect to %.128s:\n%.1024s", ldap_host,
validated_caller_input);

```

# umask

**Language:** C

**Platform:** Windows and Solaris

**Description:** The security problem occurs when a mask is passed to affect the file permissions of files created by `open()` and `mkdir()`. Resetting the mask to `0` suffices to prevent a file mode creation mask attack.

**Variants:**

**Known Types of Problems:** Privilege Escalation

**Example:**



# APPENDIX A: SMARTRISK TROUBLESHOOTING

## **I am trying to install SmartRisk Analyzer and a message is requesting an Activation Key. What do I do?**

@stake requires an Activation Key for all copies of SmartRisk Analyzer and Reviewer. Contact @stake Monday - Friday from 9 AM to 5 PM EST for your key by email [support@atstake.com](mailto:support@atstake.com) or calling 1.866.621.3500.

## **The Key I was given for another system does not work in my new system.**

Activation Keys are generated from your Machine ID, which is generated from certain pieces of information on your system. Hence, if you have changed hardware on your system, or you are trying to install SmartRisk Analyzer on a new development system, you need a new Key.

Please contact @stake for a new Activation key Monday - Friday from 9 AM to 5 PM EST for your key by email [support@atstake.com](mailto:support@atstake.com) or calling 1.866.621.3500.

## **I can't install SmartRisk Analyzer on my Linux or Solaris system.**

SmartRisk Analyzer only installs on Windows 2000, Windows XP Professional and Windows 2003 Server.

## **SmartRisk Analyzer does not load my program.**

Loading errors may occur if you have not fully compiled your application. SmartRisk Analyzer examines all binary files, and as such you are required to load a compiled version for analysis.

Compile your application and load it into SmartRisk Analyzer.

## **I received an error saying a library file is missing.**

Your application is linked to several supporting files, and since Analyzer opens and reads the binary application prior to loading to determine platform and environment compatibility, it also determines if there is a source file missing in

your application's directory. Copy the missing file and load your application into Analyzer.

### **SmartRisk Analyzer stopped in the middle of a Binary Analysis.**

Large executables require systems with faster processor speeds, more memory, and longer run times. For executables larger than 1 MB, your system should have a 2 GHz CPU and 1.5 GB RAM. Smaller systems could run out of memory before Analyzer completes.

### **SmartRisk Analyzer does not run.**

If SmartRisk Analyzer does not run, it is possible a file in the installation directory is damaged or deleted, or SmartRisk Analyzer was uninstalled. Try re-installing the program.

### **I am using the Reviewer and I can not run a Security Scan.**

You can only run Scans and Analyses on the Analyzer version of SmartRisk.

### **I ran a full security scan and no errors showed up.**

You need to run a **Binary Analysis** before running a security scan. Run a Binary Analysis by pressing **F5**, then run the scan.

### **I am trying to run software I purchased in a store through SmartRisk Analyzer, and it will not work.**

You must have complete source code of the application to run it through SmartRisk Analyzer. Purchased software like operating systems, entertainment software, or productivity tools do not include the complete source code.

### **My compiler is an older version. Does SmartRisk support my application?**

SmartRisk Analyzer only supports newer versions of Microsoft and Solaris compilers. If your application was compiled with an older version, recompile it using a newer version.

# APPENDIX B: RECOMMENDED READING

For more information on security in applications design and programming, refer to the following documents and websites.

@stake Web Site - <http://www.atstake.com>

## **Writing Secure Code, Second Edition**

by Michael Howard, David C. LeBlanc

Paperback: 650 pages

Publisher: Microsoft Press; 2nd Book and CD-ROM edition (December 4, 2002)

ISBN: 0735617228

## **Secure Coding: Principles and Practices**

by Mark G. Graff, Kenneth R. Van Wyk

Paperback: 200 pages

Publisher: O'Reilly & Associates; 1st edition (July 2003)

ISBN: 0596002424

## **Building Secure Software: How to Avoid Security Problems the Right Way**

by John Viega (Author), Gary McGraw (Author)

Hardcover: 528 pages

Publisher: Addison-Wesley Pub Co; 1st edition (September 24, 2001)

ISBN: 020172152X

## **Exploiting Software: How to Break Code**

by Greg Hoglund (Author), Gary McGraw (Author)

Paperback: 512 pages

Publisher: Pearson Higher Education; (February 17, 2004)

ISBN: 0201786958

**Secure Programming Cookbook for C and C++**

by John Viega (Author), Matt Messier (Author)

Paperback: 790 pages

Publisher: O'Reilly & Associates; (July 14, 2003)

ISBN: 0596003943

# APPENDIX C: GLOSSARY

<b>Adjusted Risk Points</b>	This value in the Summary Report gives a guideline or average security rating of the analyzed application. Adjusted Risk Points is determined by the number of severity flaws and their severity value (each severity has a numeric value) divided by the number of lines of code in the application. The lower the Adjust Risk Points value, the better the security of the analyzed application.
<b>Analyzer</b>	Analyzer runs a binary analysis, full security scan, and exports the results to a reviewer document to be read in the Smart Risk Reviewer.
<b>Annotation List</b>	SmartRisk Analyzer's Detailed Report of found security vulnerabilities, their severity, and descriptions and suggestions to fix the problems. Annotations Lists are the results of a Security Scan.
<b>Backdoors</b>	Designs in code that enable user entry into an application without logging in through the typical user interface.
<b>Binary Analysis</b>	A process SmartRisk Analyzer uses to research and evaluate all of the source code in the application that is being analyzed. It is analysis of the <i>binary</i> or compiled source code (*.exe) emulating the environment in which it was designed to be deployed.
<b>Binary Environment</b>	The libraries required by the executable in order to run properly. The user may not have, nor is he/she required to have, the source code for the <i>environment</i> .
<b>Binary Modeling</b>	Occurs during Binary Analysis. SmartRisk Analyzer creates a model of the application in order to find security flaws. Binary Modeling includes the data flow graph, the control flow graph, and range propagation.
<b>Black-Box Testing</b>	Black-box testing uses little or no information about the network to break into the network. This type of testing gives network IT professionals information about the security of their network applications. Black-box testing is usually conducted under-cover with no forewarning.
<b>Detailed Report</b>	The core output from a security scan. Detailed reports the Annotation List of each flaw and their severity, the description of the flaw, and a suggested fix for the flaw.

<b>Error (Severity Level)</b>	The second highest severity error SmartRisk Analyzer assigns to a line of code. An Error indicates major security problems with the determined line of code, and it should be fixed immediately.
<b>False Positives</b>	When performing a security scan, some analyzers detect safe code as a security vulnerability, or a false positive. SmartRisk Analyzer is designed against false positives.
<b>Five Categories of Severity</b>	The five categories of severity are in order from highest to least: Severe Error, Error, Possible Error, Warning, and Informational.
<b>Informational Alert</b>	The lowest severity ranking SmartRisk Analyzer can assign to a line of code. The discovered trigger could be problematic, or there is too little information available to determine whether the trigger is a security flaw. Manual analysis should always follow an Informational Alert.
<b>Interprocedural Data Flow Analysis</b>	SmartRisk Analyzer can analyze data as it passes through any number of function calls. It analyzes data throughout the entire program, not just within a single function.
<b>Machine ID</b>	Upon first run, SmartRisk Analyzer creates a Machine ID unique to your system. The Machine ID is a composite of several pieces of information, such as your hard drive, processor, and operating system. Use the Machine ID to obtain an Activation Key from @stake. You must enter an Activation Key in SmartRisk Analyzer before using the program. See <a href="#">page 1-2</a> for @stake contact information.
<b>Malware</b>	Software containing security flaws for malicious intent.
<b>Possible Error</b>	The third highest severity ranking SmartRisk Analyzer assigns to a line of code. The discovered trigger could create an unsafe environment. Check the code and maintain the use of precision specifiers to tighten security.
<b>Range Propagation</b>	During analysis, the SRA Engine will determine and explore all possible data values for variables in the binary. This is a time-consuming process, but necessary to create a complete and comprehensive model.
<b>Reviewer</b>	Developer review version of SmartRisk Analyzer used to read the review documents. Reviewer can not run a binary analysis or security scans.
<b>Risk Analysis Scans</b>	After the model is built, Risk Analysis Scans identify <i>trigger points</i> in the model and explore the model to determine whether the trigger points are faults in the binary, and if so the severity of the faults.
<b>Severe Error</b>	The highest severity error SmartRisk Analyzer assigns to a line of code. A Severe error indicates a serious security vulnerability, and it should be fixed immediately.
<b>SmartRisk Analyzer Engine</b>	The part of SRA that creates a multi-dimensional model of the binary. This model includes all data and control flows.

<b>Summary Report</b>	Graphical presentation detailing the number and types of security flaws in an analyzed application. Summary reports include Adjusted Risk Points.
<b>Threading/Race conditions</b>	A security scan that checks for code containing narrow openings of access vulnerable to an attack. Attackers use programs that repetitively attempt to gain access through the opening. Successful attacks occur when the opening and the attack program sync. Code written with closed access openings are less vulnerable to this type of attack.
<b>Triggers</b>	Function calls known to be problematic, and <i>may</i> cause vulnerabilities, such as the <i>printf</i> family of functions. SmartRisk Analyzer is designed to detect these problematic and possible security flaws.
<b>Unlinking</b>	During Binary Analysis, SmartRisk Analyzer breaks up the monolithic binaries into individual pieces of written code from the application, or <i>unlinks</i> them from the application. SmartRisk now has the individual functions the programmers wrote for the application. In contrast, a compiler would link these pieces together for the application. Here, they are unlinked, and each function is analyzed.
<b>Warning Alert</b>	The second lowest severity SmartRisk Analyzer assigns to a line of code. The code is functionally correct, however the triggers determine that the code can be easily attacked, and should not be trusted. The code should be checked for insecure loops and buffer boundaries.
<b>White-box Test</b>	Security testing using the entire source code of the application to find the security flaws. SmartRisk Analyzer is a form of white-box testing



# INDEX

## A

Adjusted Risk Points 2-13, 6-2  
alloc 8-19  
Analyzing output data 2-3, 2-4  
Annotation List 2-3, 2-14

## B

Binary Analysis 2-3  
Binary Modeling 5-3  
Buffer Overflow/Underflow Issues (also overruns/underruns) 2-10, 2-11, 4-11, 5-3, 5-9, 8-3, 8-19, 8-22, 8-26, 8-27

## C

calloc 8-19  
Choosing Environment Files 5-2  
Choosing Platform 5-2  
Common triggers 8-17  
Compiler environment settings 5-10  
Compiler Options 5-2  
Contact Information 1-2  
Cryptography library 2-7

## D

Degrees of severity 2-4  
Denial of Service 8-11, 8-12  
Detailed Reports 6-3  
Determining security flaw severities 7-2

## E

Editing security scan results 5-7  
Exporting results 6-3  
Exporting review documents 2-3  
Exporting security scan results 5-7

## F

fgets 8-18  
Format String Issues 8-4, 8-22, 8-28  
fprintf 8-22  
fscanf 8-26

## G

Generate a review document 6-3  
Generating a review file 4-12  
getenv 8-18  
gethostbyname 8-18  
getopt 8-18  
gets 8-18

## I

Input Validation Issues 8-18  
Installation  
    Analyzer 3-1  
Integer Overflow/Underflow Issues 8-6, 8-21

## L

Loading a binary executable 5-2

Loading source code 2-3

## M

Making review notes 4-13

malloc 8-19

measuring security results 6-2

Menu Options 2-17

Multiple errors

    different strings 7-5

    same string 7-5

## N

Note details 6-7

## O

Operating SmartRisk Analyzer 2-3

## P

Personality Settings

    Adding Creators 4-14

Prioritizing severity 7-2

## R

Reading Reports 6-4

    Detailed Report 6-5

    Summary Report 6-5

realloc 8-19

recv 8-25

Reports 2-13

    Detailed Reports 2-14

    Summary Reports 2-13

Results by Severity 6-2

Results by Type 6-2

Review documents 6-4

Risk Points 6-2

Running a Binary Analysis 2-3

Running security scans 2-3

## S

Security scan results 5-7

Security scans 5-3

    Error return checking 5-4

    Format string vulnerabilities 5-4

    Integer overflows/underflows 5-4

    Stack/Heap buffer overruns 5-3

Settings

    License Key 2-19

    Options 2-19

    Personality Settings 2-19

Severity levels 2-4, 4-11, 6-3, 7-1

    descriptions 7-2

SmartRisk Analyzer Database Document 2-2, 4-12

SmartRisk Analyzer Project

    exporting results 5-7

    saving 5-9

SmartRisk Analyzer reports 6-1

    Detailed Reports 6-3

    Summary Reports 6-1

sprintf 8-22

sscanf 8-26

Steps of operation 4-1

Sticky notes

    Creating 4-14

strcpy 8-27

Summary Reports 6-1

    Adjusted Risk Points 6-2

    Results by Severity 6-2

    Results by Type 6-2

    Risk Points 6-2

Support Platforms and Environments 2-5

    Solaris 2-5

    Windows 2-5

        Microsoft Visual C++ 6.x 2-5

        Microsoft Visual C++ 7.0 2-5

        Microsoft Visual C++ 7.1 2-6

        Microsoft Visual Studio .NET 2-5

        Microsoft Visual Studio .NET 2003 2-6

Supported Environments 1-2

syslog 8-28

System Recommendations 1-2

## T

Troubleshooting A-1

Types of Scans 2-10

## U

Using a precision specifier 2-11

## V

vfprintf 8-22  
vfscanf 8-26  
vprintf 8-22  
vscanf 8-26  
vsnprintf 8-22  
vsprintf 8-22  
vsscanf 8-26

