SecurityPrism

SECURITYPRISM WHITE PAPER

Reducing Security Risks by Ensuring Secure Application Development

GTONE
Governance Technology

Copyright © 2012 GTOne Corp. All Rights Reserved.

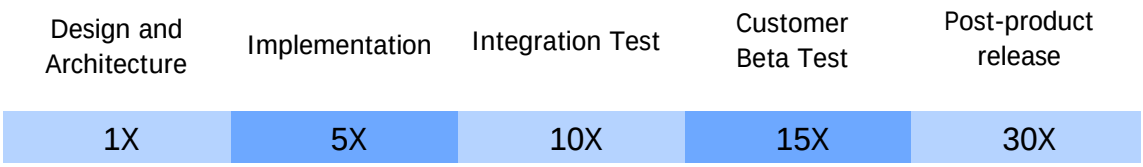
Copyright in this document is vested in GTOne Corp. The contents of the document (wholly or in part) must not be reproduced, distributed used or disclosed without the prior written permission of GTOne Corp.

Overview

Security has become one of the most important and business-relevant areas of IT. Today's increased dependency on internet makes network based software or application playing more critical roles, which has brought significant growing of malicious attacks aimed at governments, corporations, educational institutions, and individuals.

As attacks become more financially motivated and as organizations get better at securing their network, desktop and server infrastructures, there has been a shift in attacks to the application level. A critical aspect of the computer security problem is a software or application problem. It is common sense that developers always would make software defects with security holes—including bugs such as buffer overflows and design flaws such as inconsistent error handling. Malicious intruders can hack into systems by exploiting those software defects. Internet-enabled software applications present the most common security risk encountered today, with software's ever-expanding complexity and extensibility.

A study by Gartner, IBM, and The National Institute of Standards and Technology (NIST) revealed that "the cost of removing application security vulnerability during the design/development phase ranges from 30-60 times less than if removed during production. Also discoveries of security problems in source code happen late in the development life cycle or in production, which can cause significant business risk.



Source : National Institute of Standards and Technology

So, what is the most effective way to ensure secure application? Finding and fixing software vulnerabilities with automated code inspection at the early stage of development lifecycle will be one of the answers.

The static analysis tool can be used to assist with automated code inspection. It compares favorably to manual reviews, but they can be done faster and more efficiently. The tool also encapsulates deep knowledge of underlying rules and semantics required to perform this type of analysis such that it does not require the human code reviewer to have the same level of expertise as an expert human auditor.

Its aim is automatically detecting and locating defects in source code. Those defects can be broadly divided into two categories; security vulnerability and quality. The vulnerability is a weakness which allows an attacker to attack a system, decreasing system's security assurance. The defects associated with software quality may vary ranging from potential errors and bad performance factors to non-compliance with development standards. In this paper, we will focus on the security vulnerability.

How to prevent security vulnerability

Corporations utilize a variety of application security tools such as protocol testing, application firewall and web application scanning according to changes in hacking trend. However, it's impossible to block malicious cyber-attack 100% with the existing application security tools. Because recent cyber-attack becomes more intelligent beyond security policy of corporations and government organizations, application can be exposed to hackers' attack if application source code is insecure in spite of security tools.

Security may well be vulnerable since application is developed based on incomplete security methodology and it's difficult to prevent intelligent cyber-attack with only general application security tools.

Also, attack patterns have been recently developed to attack not only application server security vulnerability, but also application logical error. Corporations utilize a variety of application security tools such as application firewall or web scanner to block attacks.

A various of ways of ensuring security

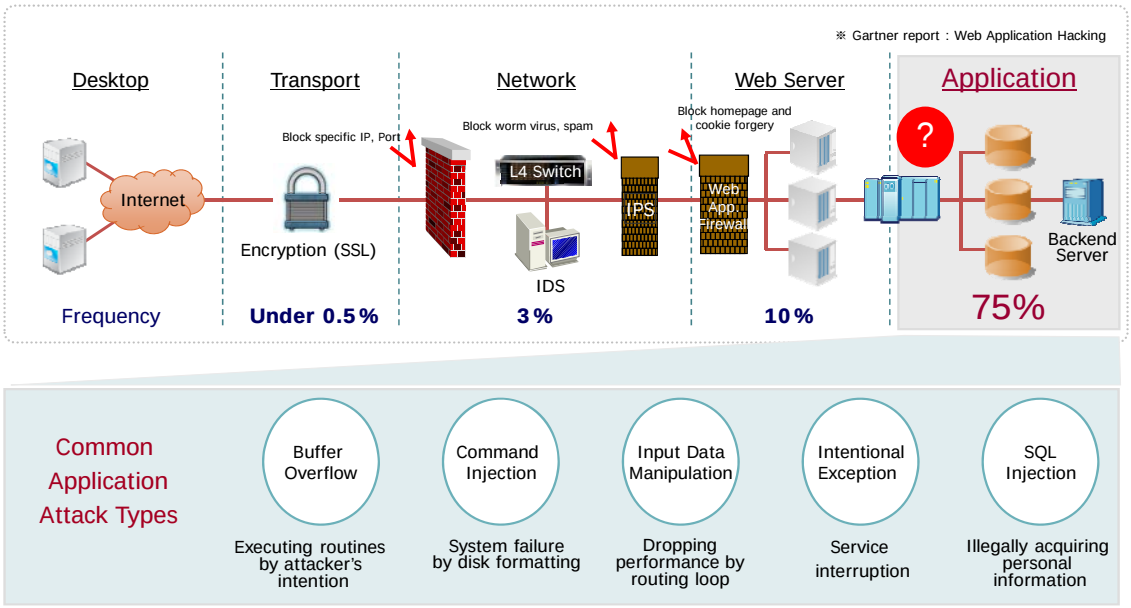
First, the most common way of ensuring application security, application firewall blocks various attack efficiently, preventing HTTP (80) protocol attack in network layer 7 (application level). However, firewall configuration is complicated and causes server performance degradation.

Second, a web application security scanner is a program which communicates with a web application through the web front-end in order to identify potential security vulnerabilities in the web application. It's efficient for inspecting security vulnerability through periodic inspection, as inspecting vulnerability after mock attack to WAS and web server. However, there are too much pages which scanner can't inspect according to link structure and web development technologies and too many parameters to check and input items to enter. Plenty of time and human resources are required for scanning and patching. Also, since web application scanner doesn't understand inner mechanism of web application, it gives only general recommendation. Therefore, developers have a hard time to find in which they should modify source code.

Finally, static analysis tool analyzes source code's vulnerability which brings fundamental problem of application security. It directly provides professional and applicable source code modification guideline to developers. If you modify one source code, it can remove vulnerability from several pages. It's because one source code has normally relations with several source codes. Furthermore, as developers can see inner working mechanism of application, they can easily find vulnerability which other tools can't find.

Changes in perception about source code vulnerability inspection

It recently happens a lot to attack vulnerability shown in application development because it is relatively easy to detect and attack web application vulnerability and insecure programs are used even since development phase.



[Figure 1] Occurrence rate of security violation and hacking attacks

A Gartner study reveals that 75% of cyber-attacks are done at the web application level and 90% of websites are vulnerable to attack. In other words, hacking methods have been changed from attacking system and network vulnerabilities to using application source code vulnerability, it's required to prevent hacking from the early stage of application development.

As application security issues becomes big issues globally, new types of organizations currently appear and provide practical and efficient information about source code security.

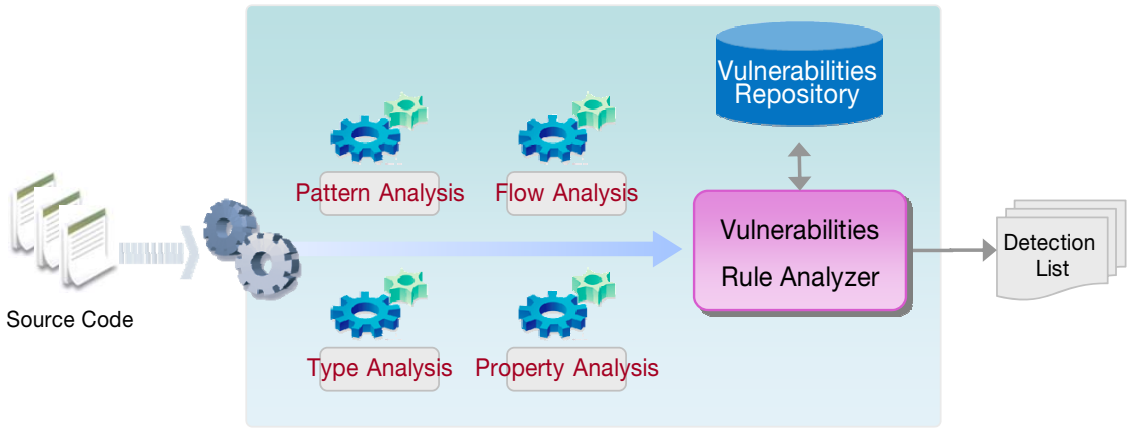
As CWE (Common Weakness Enumeration), OWASP (The Open Web Application Security Project) and CAPEC (Common Attack Pattern Enumeration and Classification) are the outstanding communities, they provide application vulnerabilities, application security standards & coding guidelines and attack methods. Source code security issues that they provide are very important and they have changed awareness of application security threat.

What is source code vulnerability inspection tool?

Source code vulnerability inspection tool is to detect source code exposed to hackers' attack and bring more complete security from application development phase. Through analyzing application's source code without running it, you can track vulnerable source code in development phase, modify vulnerable source code before production phase and remove security vulnerability.

Generally there are 2 types of analysis for applications. First, static analysis is to analyze all cases possible in source code. You can relatively detect most of vulnerabilities due to wide analysis range. Second, dynamic analysis is to run program practically and analyze vulnerability after finishing development. It takes plenty of time due to program running and is hard to perceive whether to check all the cases.

In this paper, we focus on static analysis which provides a variety of vulnerability information without running program and finds source code error in the early stage of development process.



[Figure 2] Basic flow of Source Code Static Analysis

A variety of technology is applied for static analysis of source code. Security vulnerability analyzer provides pattern, type, flow and property analysis. The pattern analysis is to detect vulnerabilities by code shape and usage pattern. The type analysis is to detect vulnerabilities through precise calculation of language's unique types such as variable, value, expressions, etc. The flow analysis is to detect vulnerable source codes through examining control flow and data flow. The property analysis is to detect vulnerabilities by considering language specific properties.

Benefits of the tool

As application function and size are getting bigger, number of source code line is extremely getting increased. It causes to increase time and cost. There is a limit for developers to remove source code vulnerability. U.S. department of defense and software engineering institute reveal that no matter how experienced professionals are, they can only check 1,000 lines a day for detecting errors. Automatic source code analysis tool is required strongly to solve this problem.

You can expect benefits below through analyzing source code vulnerability automatically.

- Reduce development time, detecting in the early stage of development through security defects detection
- Automatically analyze source code in a short time which too much time and human resources are required for

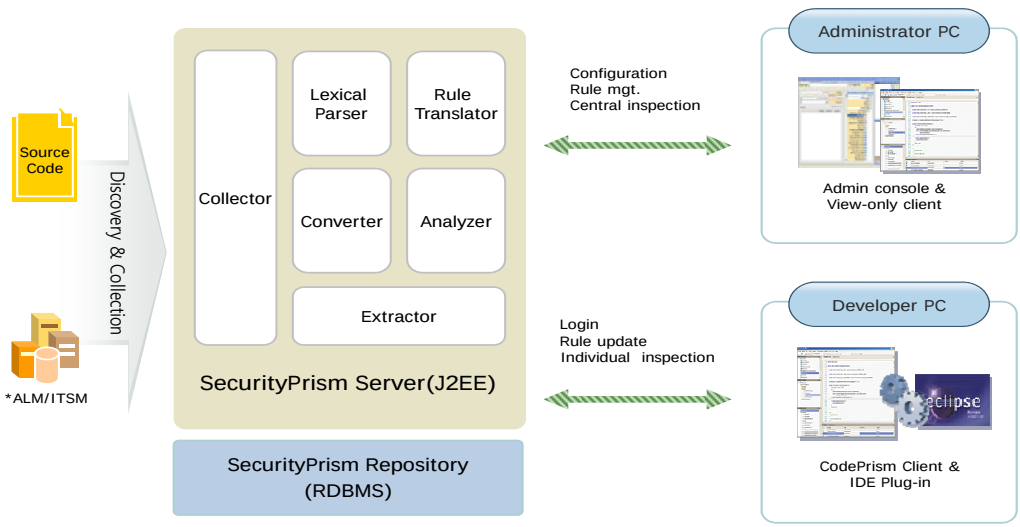
It was recognized in the past that system manager maintains application security. However, developers and business users can strengthen vulnerability inspection to remove security weaknesses through static analysis at the application development phase and constant monitoring system of application security vulnerability can be established. Organization can not only provide secure service through improving application security, but improve their business competitiveness through strengthening security and reliability.

SecurityPrism

What is SecurityPrism?

SecurityPrism is a source code vulnerability inspection solution to detect software security vulnerability in the early stage of development lifecycle. Without configuring compiler environment or running programs, it exactly locates code lines which violate the pre-defined security rules.

It provides secure coding guidelines based on international standards such as CWE and OWASP and helps to modify source code correctly. Developers and QA can prevent a tremendous disaster by malicious hacker's attack through removing defects in advance and improve software security.



[Figure 3] SecurityPrism Architecture

SecurityPrism server is based on J2EE technology to provide platform independence and stability. SecurityPrism provides a variety of functions like below:

- Collector: collects source file communicating with remote/local computer or configuration management server which has application source codes for analysis
- Lexical parser: generates parsing tree for input program source file, which is used to figure out program structure and lexical errors etc.
- Converter: converts source code to common intermediate code that is input to analyzer.
- Rule translator: provides information required for analysis, interpreting security vulnerability rules created with GTOne's unique Rule Description Language
- Analyzer: inspects source codes to detect violations of pre-defined rules, referring to some information from the rule translator
- Extractor: saves data extracted through analyzer, into central repository
- Repository: archives all data analyzed and provides useful information to developers and administrators as application knowledgebase (RDBMS used)

For the client side, an administrator and developer can access to the SecurityPrism Server from their own PC:

- SecurityPrism Admin Console: enables administrator to manage the tool's configurations and rules using internet protocol.
- SecurityPrism View Client: enables users to view the analysis results on SecurityPrism Server. It also provides rule descriptions and various reports.
- SecurityPrism Client: it is a stand-alone program making developers to be able to download vulnerability rules from server and analyze their source files.
- Eclipse Plug-In: makes developers to be able to download vulnerability rules from server and analyze their source files directly through Eclipse IDE.

What types of security defects can be detected?

In this section, we will go through some examples of security vulnerabilities which can be identified using the static analysis tool. Although many more types of defects can be detected by SecurityPrism, the examples in this section are just for the reader's better understanding.

SQL Injection

One of the major hacking methods is the SQL injection attack. Such attacks exploit security vulnerabilities and insert malicious code (in this case script tags) into the database running a site. When user input, for instance via a web form, is not correctly filtered or checked, the code peppers the database with malicious instructions. Recovery can be difficult, and there are numerous cases of website owners cleaning up their database only to be hit again a few hours later.

Let's take a simple code example.

```
1001        Statement stmt = conn.ceateStatement();
1002        String name = request.getParameter("name");
1003        String    query    =    "SELECT    *    FROM    employees    WHERE
1004        username='"+name+"'";
...         ResultSet rs = stmt.executeQuery(query);
            ...
```

At the line number 1002, an external input value is assigned to a variable, name. It is directly used in the line number 1003 to build a dynamic SQL statement. The SQL is executed by JDBC API for dynamic query to get data result set from target database. If the 'name' variable would contain malicious string (' OR 1=1--), the SQL statement will return the list of all employees.

Therefore, the above sample should be modified as below:

```
1001          Statement stmt = conn.ceateStatement();
1002          String name = request.getParameter("name");
1003          if ( name == NULL ) return;
1004          // check if the 'name' contains malicious code (' , ; ,--)
1005          if ( name.indexOf(" or =" ) != -1 || name.indexOf(" ;" ) !=
1006          -1 || name.indexOf(" --" ) != -1 ) return;
1007          String query = "SELECT * FROM employees WHERE username=?";
1008          // Use static SQL Query API
1009          PreparedStatement pstmt = conn.prepareStatement(query);
1010          pstmt.setString(1,name);
...           ResultSet rs = pstmt.executeQuery();
              ...
```

The line number 1004 checks if the 'name' variable contains malicious string or not. Then, the variable 'name' is not directly used in SQL statement, instead, parameterized query is built in the line number 10066. Also the query is executed by JDBC API for static query. In the above fixed secure source code, an attacker cannot gain some information from the database because the SQL statement won't be built as wanted.

Cross-Site Scripting (XSS)

Cross-Site Scripting attacks are a type of injection problem, in which malicious scripts are injected into the trusted web sites. Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site. These scripts can even rewrite the content of the HTML page.

Let's take a simple code example:

```
1001          String name = request.getParameter("name");
1002          ...
1003          if (param != null) {
1004          out.println("Name:"+ name);
1005          }
```

If the variable 'name' contains malicious java script code, for example, <script>alert(document.cookie);</script> by external user input, cookie data can be transmitted to the attacker. Therefore, developers always must check input value if it contains special characters for malicious script code.

Here is the secure coding example:

```
1001          String name = request.getParameter("name");
1002          ...
1003          // whitelist check for malicious input (<, ", ', &, >)
1004          if (name != null) {
1005          name = name.replaceAll("&", "&amp");
1006          name = name.replaceAll("<", "&lt;");
1007          name = name.replaceAll(">", "&gt;");
1008          name = name.replaceAll(" " " " , "&quot;");
1009          name = name.replaceAll(" '" , "&apos;");
1010          out.println("Name:" + name);
1011          }
```

It is recommended not to allow non-predefined input values by checking input strings in source code.

Command Injection

The purpose of the command injection attack is to inject and execute commands specified by the attacker in the vulnerable application. In situation like this, the application, which executes unwanted system commands, is like a pseudo system shell, and the attacker may use it as any authorized system user. However, commands are executed with the same privileges and environment as the application has. Command injection attacks are possible in most cases because of lack of correct input data validation, which can be manipulated by the attacker (forms, cookies, HTTP headers etc.).

Let's take a simple code example.

```
1001          String command = request.getParameter("command");
1002          ...
1003          if (command != null) {
1004          Runtime.getRuntime().exec(command);
1005          }
```

For the above sample code, If attackers can input some OS commands such as “rm”, “format” and “passwd”, the system can be damaged significantly.

To ensure secure coding, developers must eliminate special characters which can affect OS command structure (ex: (;, >, &, |), from external inputs. If the application should execute some parameterized OS commands from external input, developers must be recommended to provide safe commands as options allowing users to select one of them. They also have to always check if the input commands are from whitelist.

The below code example doesn’t allow executing bad commands:

```
1001          String command = request.getParameter("command");
1002          // check the input value for a command (; ,>, &, |)
1003          if ( command .indexOf(" ;" ) != -1 ||
                command .indexOf(">" ) != -1 ||
                command .indexOf("&" ) != -1 ||
                command .indexOf("|" ) != -1 ) return;
1004          // use allowed commands list
1005          String allowCommand[] =
              {" command1" ," command2" ," command3" };
1006
1007          ArrayList arr = new ArrayList();
1008          for ( int i=0;i<allowCommand.length;i++ )
1009          arr.add(allowCommand[i]);
1010
1011          // deny executing non-whitelist commands
1012          if ( !arr.contains(command) )
              throw new MyException("Error!!");
1013          Runtime.getRuntime().exec(command);
```

Even though attackers manipulate command string, non allowed commands cannot be executed through the above program.

Information exposure through an error message

The vulnerable application would generate an error message that includes sensitive information about its environment, users, or associated data.

The sensitive information may be valuable information on its own (such as a password), or it may be useful for launching other, more deadly attacks. If an attack fails, an attacker may use error information provided by the server to launch another more focused attack. For example, an attempt to exploit a path traversal weakness might yield the full pathname of the installed application. In turn, this could be used to select the proper number of ".." sequences to navigate to the targeted file. An attack using SQL injection might not initially succeed, but an error message could reveal the malformed query, which would expose query logic and possibly even passwords or other sensitive information used within the query.

Let's take a simple code example:

```
1001          try {
1002          ...
1003          PreparedStatement p = conn.prepareStatement(
1004          "SELECT * FROM userTable WHERE Name = ?");
1005          ...
1006          }
1007          catch (Exception e) {
1008          e.printStackTrace();
1009          throw e;
1010          }
```

The catch block in the above sample is handling SQL related exception. In this case, the error message can contain sensitive information such as SQL statement's structure or system information. If this error output is included in http response, the sensitive error stack information would be displayed in a web page.

Developers must make attackers not to be able to guess what type of error occurred.

```
1001          try {
1002          ...
1003          PreparedStatement p = con.prepareStatement(
1004          "SELECT * FROM userTable WHERE Name = ?");
1005          ...
1006          }
1007          catch (SQLException sqle) {
1008          // do not use default JDK exception
1009          // use user defined Exception
1010          throw new MyException("Error! Please try again!" );
              }
```


Buffer Overflow

In computer security and programming, a buffer overflow is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory. This is a special case of violation of memory safety.

Buffer overflows can be triggered by inputs that are designed to execute code, or alter the way the program operates. This may result in erratic program behavior, including memory access errors, incorrect results, a crash, or a breach of system security. They are thus the basis of many software vulnerabilities and can be maliciously exploited.

Programming languages commonly associated with buffer overflows include C and C++, which provide no built-in protection against accessing or overwriting data in any part of memory and do not automatically check that data written to an array (the built-in buffer type) is within the boundaries of that array.

Let's take a simple code example:

```
1001          void manipulate_string(char* string){
1002          char buf[24];
1003          strcpy(buf, string);
1004          ...
1005          }
```

A parameter string is copied to a buffer variable without checking size. This programming pattern would result in buffer overflow. Therefore, the source code should be fixed as below:

```
1001          void manipulate_string(char* string){
1002          char buf[24];
1003          if (strlen(string) < sizeof(buf))
1004          /* check the size of both parties */
1005          strncpy(buf,string,sizeof(buf)-1);
1006          /* use strcopy and  sizeof() method */
1007          buf[sizeof(buf)-1] = '\0';
1008          /* string should be terminated with null character */
1009          ...
1010          }
```

Here is another simple example:

```
1001          char buf[50];
1002          char *ptr;
1003          int i=0;
1004          for ( i=0; i<= sizeof(buf); i++){
1005          *ptr=&buf[i];
1006          /* refer to a character, 51th idex in buf string */
              }
```

The line number 1005 refers to out of index memory, buf[50]. It would be result in buffer overflow. Because bounds checking can prevent buffer overflows, it should be fixed as below:

```
1001          char buf[50];
1002          char *ptr;
1003          int i=0;
1004          for ( i=0;I < sizeof(buf) ;i++){
1005          *ptr=&buf[i];
1006          /* refer to a character, 51th idex in buf string */
              }
```

Conclusion

There is no magic bullet to prevent a cyber attack. The technologies themselves are not universal panaceas, even when the vulnerabilities have been dealt with. However, organizations, at least, have to make an effort to reduce security risks due to vulnerable applications. Ensuring development of secure application from the development phase is a major step for the goal.

How can you ensure development of secure application? You need to provide developers with secure coding standards and check the compliance of them before finishing development. The best way to do this is applying static vulnerability analysis tool such as SecurityPrism.

SecurityPrism can detect source code exposed to hackers' attack and bring more complete security in the early stage of application development life cycle. Furthermore, by the optional license, SecurityPrism enables you to conduct source code's both vulnerability and quality inspection at once. It means that you can conveniently check your application's security defects and quality including potential errors and performance issues in a single environment.

Finally, SecurityPrism will help you:

- Reduce huge business risks and costs triggered by application security incidents.
- Catch application's security weaknesses early
- Improve application security and quality with productive way

http://www.gtonesoft.com

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the GTOne License Agreement and may be used only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied photocopied, reproduced, translated, or reduced to any electronics medium or machine readable form without prior consent, in writing, from GTOne, Corp.

Information in this document is subject to change without notice and does not represent a commitment on the part of GTOne. The software and documentation are provided “as is” without warranty of any kind including without limitation, any warranty of merchantability or fitness for a particular purpose. Future, GTOne does not warrant, guarantee, or make any representations regarding the use, or the results of the use, of the software or written material in terms of correctness, accuracy, reliability, or otherwise.