



The Security Bug Catcher

reloaded, part two

Andreas Rüegg

SSC
Semester Project
February 2004

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Dr. Philippe Oechslin
EPFL / LASEC

LASEC

Table of contents

I. Introduction	4
II. Goals of this second part	5
a. Definition of the goals at the project start	5
b. Goals set during project for further enhancements	6
III. Program Architecture	8
a. GUI	8
b. Explorer	8
c. DottyFileWriter	11
d. Statistics and error-log writer	13
e. State	15
IV. Basic extensions and modifications of the program	16
a. GUI for controlling	16
b. Graph visualization	17
c. Strategy to limit number of server queries (minimal tree / range)	20
d. Specialized exploration strategies	20
e. New protocol implementation (new file structure); How to?	21
f. Indexing and grouping of syntax strings and commands	24
g. Drawing modes for graph visualization	24
h. Performance enhancement	24
V. Smaller extensions and modifications	26
a. Structured error-logs	26
b. Possible continuation on server crash	27
c. Visualization during exploration	28
d. Removal of error with id 2	28
e. Modification of error logs	28
f. Error log summary writer	29
VI. Some implementation challenges	31
a. Fundamental disregards in server's protocol implementations	31
b. Timeout and server closing recognition	31
c. Graph dimensions	32
d. Treating multi-line replies	32

VII. FTP Server examination reports	34
setting the context	34
a. BisonFTP Server V4R1 (Windows)	35
b. oftpd 0.3.6-5.1 (Linux)	36
c. Broker 6.1.0.0 (Windows)	36
d. Serv-U 5.0 (Windows)	36
e. Ability FTP Server (Windows)	37
f. wu-ftp 2.6.2 (Linux)	37
g. Pure-FTPd 1.0.17 (Linux)	37
h. vsFTPd 1.2.1 (Linux)	37
i. Microsoft FTP Server (Windows)	37
j. WS_FTP Server 4.02 (Windows)	37
k. RaidenFTPD 2.4 (Windows)	37
l. Black Moon FTP Server 2.8 (Windows)	38
m. ArGoSoft FTP Server 1.4.1.3(Windows)	38
VIII. POP3 Server examination reports	39
setting the context	39
a. GoPOP (Windows)	39
b. Inframail 6.17 (Windows)	40
c. teapop 0.3.5 (Linux)	40
d. Solid POP3 Server (Linux)	40
e. popa3d (Linux)	42
f. mailutils-pop3d (Linux)	42
g. TmPopAdmin 2.13 (Windows)	42
IX. Conclusion	43
a. Conclusion of the server test series	43
b. Conclusion of the entire project	44
X. Appendix	45
a. Dotty / Graphviz	45
b. RFC's	45
c. Code of the entire 'explore' method	45

I. Introduction

The Security Bug Catcher was initiated by Philippe Oechslin of the LASEC at the EPFL. This project was from its basic concepts thought to be continued, extended and improved during several semester projects. The step documented here is the second semester project concerning this program. In a first step by Jan-Olivier Fillols the basic framework was set and developed. The program obtained was able to query an FTP server (just its control connection part was already implemented) with commands and syntax string defined in the protocol definitions given as files to the program. Starting from the first state it tested all possible combinations of commands and accompanying syntax strings until it reached a defineable depth and reported the errors in log files, divided in 6 different kinds of errors. Besides several protocol implementation faults, the program found buffer overflows in two server applications which crashed after sending a specific command-string combination.

For the winter semester 2003/2004 this project continuation was proposed by the LASEC and I was fortunate enough to get it as semester project in my 5th semester of communication systems. The project treats several subjects that I'm interested in, at the same time. First of all communication over internet and it's protocols is a very basic application whose combination with security aspects is not only very interesting, but also very important in our world, where much confidential information is transferred via internet. Secondly I like much Java as programming language and this project allowed me to work a lot on it and also improve my knowledge concerning Java.

In the first place this project gives information to this second part I've realized this semester. Compared to the first part report, this one gives apart from information of this semesters work also a technical reference for further programming on The Security Bug Catcher (see section III).

II. Goals of this second part

II.a. Definition of the goals at the project start

The Introduction already gave a very brief summary of what has been done as first step of this project. For more details please consult the report of Jan-Olivier Fillols which describes the first part.

At the beginning of this second part of The Security Bug Catcher, for me the first thing was the study of the code developed till then. This work was quite dry and took the first two weeks of the semester. In these two weeks we also roughly set the goals that are listed below. But in the whole project I had the option to add other features and change other program parts depending on the project development. Ideas from my part were very welcome, have been discussed with Philippe Oechslin and had big influence as one can see in sections IV and V.

Overview of the starting goals:

- Implement exploration strategies for:
 1. reducing the amount of queries (minimal tree / range)
 2. continue after an error
- Implement entire FTP protocol and an other one to show the usage of the program with any protocol
- visualize the tree of exploration
- make structure of input-files for syntax strings more efficient

The first point is a very important one where much of work has been spent on. A big limitation of the program developed in the first part was the large number of polls the program executed, because it tried all possible combinations until it reached a certain depth defined by the user. One can imagine, that even only with the control connection implementation of FTP which encloses about five commands (USER, PASS, ACCT, REIN, QUIT) and an average number of five syntax strings for every command, the amount of commands sent to the server to reach depth three (= to visit every state at least once, because the control connection implementation contains three different states) was somewhere in the order of $(5(\text{commands}) * 5(\text{strings}))^3(\text{depth}) \approx 15'000$. At the speed the program made the server pollings at that time, an entire exploration of depth three could have taken more than an hour. It was imperative to change this limitation because of the second point already mentioned, the fact that another project goal was to implement the entire FTP protocol which contains about 30 commands and three more states (see section IV.e.). Considering that the amount of commands increases exponentially with the depth, it was evident that the number of polls had to be limited by a strategy that still allows to find the most of server errors.

At this point the idea was to define what we call a minimal tree in the state tree of the pollings and allow the user to choose a range that makes the program not go away further from the minimal tree than the number of nodes defined with the range. Details on that system are showed in section IV.c.

The second kind of strategies considered continuation after an error. This was an essentially experimental task I liked to implement because we didn't know at this point if these strategies would really lead to detect other server vulnerabilities. The finally implemented strategies are closer described in section IV.d.

As mentioned above, the second part consists of writing the whole FTP protocol in the syntax taken by the program as input. In addition to that it was requested to implement

a totally new protocol to verify if the usability with other protocols is supported and to make changes to the program code if there should be problems (see section IV.e.).

A third main part was the development of a clear visualization of the exploration done by the program. The idea was to generate the tree of visited states that shows very clearly the behaviour of the server and the errors it produces. All details concerning this implementation are shown in section IV.b.

Finally, the program presented after the first step had an inconvenience with the syntax string entered in files. There existed one file for every command of the protocol and each file contained syntax strings like this: "USER %s%s%s%s%s%s". The fundamental change was to split the commands from the strings themselves and to create groups of strings which can afterwards be applied on every command. This makes the whole syntax definitions clearer and easier to handle and modify. In addition to that all the strings are stored in one file and each string only has to be defined once to be applied on any desired command. Further details concerning this enhancement can be found in section IV.f.

II.b. Goals set during project for further enhancements

Because of the freedom I had in the whole project, there was always the possibility to discuss things and set new goals or modify existing ones. The most important ones of this category I'd like to mention here:

- Creation of an extended GUI to provide the user controlling the exploration and setting it's parameters
- Different drawing modes for the graph visualization
- Performance increasing, managing server timeouts and recognizing server crash
- Continuation on a server crash, to find possible other errors in further exploration

The program written in the first part already possessed a GUI to show in real-time the exploration results and the progress of the exploration. The parameters such as server address, server port, exploration mode and exploration depth were given to the program as arguments at starting from the console. The implementation of several new features that allows the user to set plenty of parameters made it necessary to enlarge the GUI and develop it from a purely informational one to a combination of informational and operational GUI. In other words there was not really a new project goal set, but a necessity identified for the further development and growing of the project. First, this new GUI makes the handling of the program much easier and secondly there is no need to enter every time all the parameters manually, because often there are used default values for several features. The new GUI and its usability is presented in section IV.a.

Once the doty graph drawing module was built, we had to make the experience, that this new visualization tool was very nice, but not yet as useful as we had imagined before. The amount of nodes created at larger explorations was so big that it could not even be shown entirely by doty and that the overview suffered very much from this huge tree. At this point we set a new project goal to introduce a feature that enables drawing not of the entire tree, but only of the branches where an error occurred. The

whole graph visualization generated by the program in this mode became much clearer than with the entire tree. The introduced graph drawing modes are shown in section IV.g.

One day I realized that the response of the server was much faster than the flow of the program and I decided to search the code for parts that brake the program. This work was quite difficult and included a lot of code analysis. Finally I found ways to speed up the program which allows now to do explorations in much less time than before. Problems I've been faced with and the solutions developed to overcome them are shown in section IV.h.

As last example I want to come to the fourth point mentioned above. At the moment when the program crashed a first server implementation (it produced a buffer overflow in the POP3 server GoPOP which is shown in section VIII.a.) we came to the conclusion that it would be helpful to have an implementation that warns the user in case of a server crash and stops the program (not abort, just halt). The user now has time to restart the server and afterwards tell this the program which continues the exploration at the point the server crashed. This way all the implementation faults of the server can be explored (see section V.b.).

Beside these goals that have been set during the project there have been much more smaller items or small bugs from the first part that had to be fixed. Some of them are mentioned in separate sections, others are not mentioned at all or just in passing because they are not of big interest or had for example only internal influence on the program code.

III. Program Architecture

III.a. GUI

The GUI is the graphic interface that permits the user controlling the program (start, stop abort, visualize) and setting the exploration and visualization parameters. Its frame is created by the *'main()'* method situated in the *'Launch'* class by instantiating the class *'MyFrame'*. *'MyFrame'* afterwards arranges all the Buttons, TextFields, Choices and Checkboxes and adds the ActionListeners. Until here the main program hasn't been started, yet. Once the user has entered the desired parameters and clicks the start button, an instance of the *'Explorer'* class is created, all the parameters are passed from the GUI to this instance and its thread is started. The *'run'* method of the explorer now leads the whole program which is shown more closely in section III.b.

During exploration the results at this point of execution can already be visualised by clicking the Visualize button. The method *'concatFiles'* in the *'DottyFileWriter'* class (see section III.c.) is called to build the final DOT-file.

A running exploration can be stopped with the Stop button and the results (graph and error logs) obtained till now are stored. This button stops the whole program, destroys all the instances (stops all the running threads) created at starting to provide a clean restart without any predefined variables and objects.

III.b. Explorer

The *'Explorer'* class contains the heart of the program. As mentioned in section III.a. an instance of *'Explorer'* is created in the *'MyFrame'* class and *'run'* is called by starting a new thread. The *'run'* method, after the instantiation of necessary classes, namely *'State'* (see section III.e.), *'DottyFileWrite'r'* (see section III.c.), *'Statistics'* (see section III.d.), starts with the exploration by calling the method *'explore'* with the root state as argument.

For better understanding I won't describe the whole *'explore'* method in words, but show at this point a simplified pseudo-code version of this method (see beginning of next page). Remark: The entire code of the *'explore'* method is given in the Appendix (X.c.).

The key element of this method is its recursive calling. As we will see just below, the method *'goToNextState'* calls recursively the *'explorer'*. This way it is much more efficient to do the tree-like state transitions (look also at Figure 1 that shows in a state diagram the program execution).

The very short version of *'explore'* that is shown below in pseudo-code only includes the fundamental methods and pieces of code concerning the iterations for the exploration. In reality this method takes about 300 lines of code because here I didn't mention several features used for drawing, strategies and other functionality.

To really understand the pseudo-code written below I'll also give short explanations on the methods *'stopIteration'*, *'goBackToPrecedentState'* and *'goToNextState'* which are all defined in the *'Explorer'* class. The method *'storeError'* is not very important for the understanding of the *'explorer'*, one just has to know that it gives an error to the *'Statistics'* class where it is treated for later writing into the log files.

```

explore(State s) {
  if (!stopIteration(stateName)) {
    for each command {
      for each syntax {
        send command with syntax;
        get answer and treat timeout;
        if (timeout) {
          decide on error_id;
          stats.storeError(syntax, error_id, pastCommands,...);
          goBackToPrecedentState(state,command,reply_code,...);
          continue;
        }
        get reply_code;
        get next state;
        if (next state == "unknown") {
          stats.storeError(syntax, error_id, pastCommands,...);
          goBackToPrecedentState(state,command,reply_code,...);
        }
        else {
          stats.storeError(syntax, error_id, pastCommands,...);
          goToNextState(thisState, nextState, command,...);
        }
      }
    }
  }
  // here is a goBack without drawing the node because this is just a
  // virtual iteration (no commands are sent to server)
  goBackToPrecedentState(state,command,reply_code,...);
}

```

stopIteration: This method contains the iteration break conditions. It checks for depth, range, exit-state and other strategy specific break conditions. If this method returns true, there is no action in the 'explore' method, but a jump to the end, what produces a return to the precedent state ('goBackToPrecedentState') without having sent something to the server, though this node is "virtual" and won't be drawn in the tree.

goBackToPrecedentState: Here I'll explain the functioning of this method but without considering the cases of the several strategies. These special behaviours can be understood by reading section IV.d. which is a less technical description but in my eyes sufficient for the understanding of the code.

There are two different applications of the 'goBackToPrecedentState' method. Either at the end of a recursion (if the 'explorer' method reaches its end (after a 'stopIteration()' = true)), or in case of an error where the exploration is continued with the next syntax/command which is done by calling the continue statement after the 'goBackToPrecedentState' method. These two applications are indicated by the 'writeThisNode' boolean given as argument to 'goBackToPrecedentState'.

Apart from the handling of drawing a node or not its work is to close the server connection and resend all the commands which are stored in the Vector so that a new command can be sent to the server.

goToNextState: The main task of 'goToNextState' consists in introducing a new iteration of explore with the 'nextState' found during the exploration of the present command.

Same as 'goBackToPrecedentState' it also handles the drawing of the next node which in this case can be or not in the 'mainTree' (minimal tree) and therefore method 'isInMainTree' is called (details on this strategy are found in section IV.c.).

Here it is not necessary to resend all the commands executed before, because in the new explorer-recursion the directly following command will be sent.

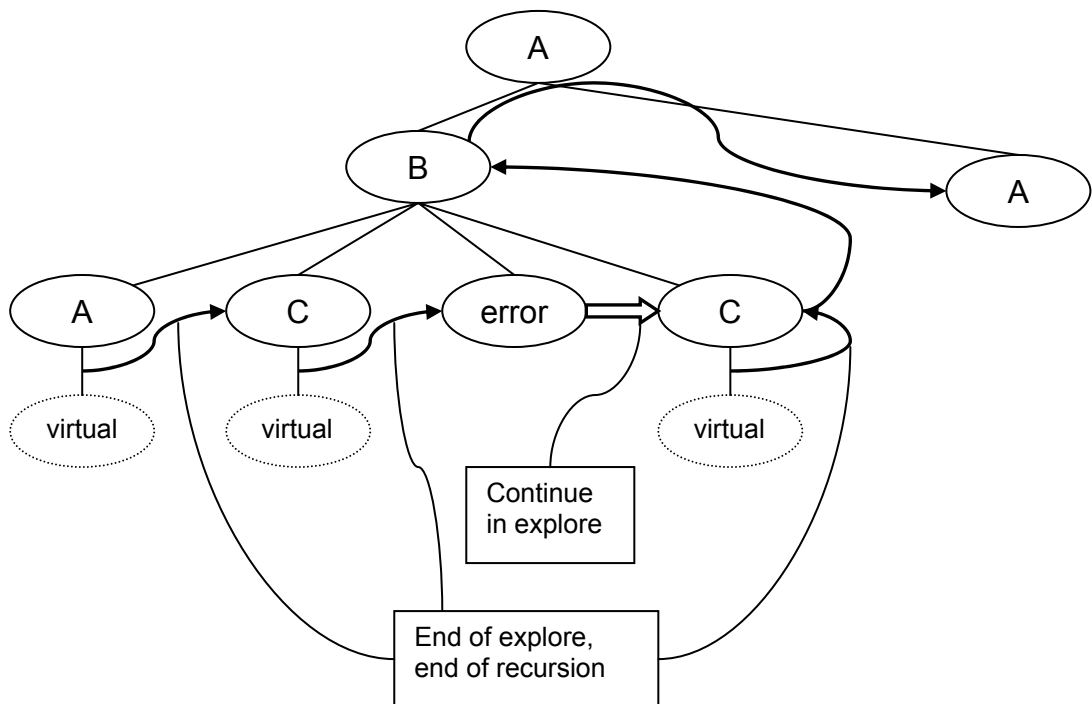


Figure 1: state diagram of exploration

During execution of the 'explore' method at several points of the code the differentiation of an eventual error takes place. At this point I'll give a summary of the different errors treated by this program and add a brief explanation:

Id	Timeout	Can connect	explanations
1	yes	yes	Server doesn't respond but connection is alive
3	yes	no	Server crashed
4	no	yes	Connection is closed while awaiting reply but before timeout.
5	no	yes	A reply like this one is not defined after given state and sent command.
6	no	yes	Transition to this state should not be possible referring to server_settings file.

Finally I'll give some explanations concerning the most important variables used in the class 'Explorer':

Vector commands: Contains all possible commands of this protocol (read from "commandSyntax" file)

Vector states: Vector filled with all the 'State' instances.

String[] syntaxArray: Contains all the syntax strings defined in the syntaxStrings file (the number in front of the string is the index where it is placed in the 'syntaxArray'.

Hashtable idsOfCommands: Contains keys of each command with an array of syntax ids used by this command.

Vector executedCommands: Contains strings of this pattern: "command syntax_id" , for every executed command from the root to the current state.

LinkedList visitedStates: Ordered Vectors for every state from the root to the current node with information for drawing:
[boolean painted,String nextState, String command, Integer syntax, String reply_code, Integer color]

HashTable reexploreArchive: Table of states (keys) with concerning 2-dim. array for sent command-index / syntax-index. The array counts appearance of each command-syntax combination sent from a specific state (to decide if a state is reexplored later if the user chose the reexplore-strategy)

LinkedList statesInMainTree: Vectors for each element in minimal tree. Information in every vector: (State, stateNumber)

III.c. DottyFileWriter

The 'DottyFileWriter' class was added as a new item in this second part as basis for the tree-visualization. This section covers the technical coding part. More information on the background of the tree-visualization is given in section IV.b.

In the running program this class is instantiated by the 'Explorer' class. A thread is created so that writing the graph files happens asynchronously in a separate thread. Because of that there is a waiting list for nodes that have to be written. This LinkedList 'toWrite' is accessed by a FIFO (first in first out) algorithm.

To enter a node into the 'toWrite' list the caller can use the public 'setArgs' method:

```
public void setArgs(String node1, String node2, String edge, Integer
                    color) {
    Vector temp = new Vector();
    temp.addElement(node1); // node before
    temp.addElement(node2); // new node
    temp.addElement(edge);  // new edge
    temp.addElement(color); // color
    towrite.addLast(temp);
}
```

New nodes to write are added at the end of the 'toWrite' list. In the 'run' method there is checked if 'toWrite' is not empty and then the first element is removed for treatment. The calls of 'writeNode' and 'write Edge' append the settings each to a separate file. By

calling the public method `concatFiles`, the writing process in `run` is halted by the boolean `breakLoop` to avoid reading and writing at the same time and the two files created by `writeNode` and `write Edge` are put together in a file called "graph.txt" that is stored in the corresponding error log folder. After this call the process goes on by setting `breakLoop` to true. Due to that, anytime during exploration (by clicking the "visualize" button) the "graph.txt" file can be written by continuing the exploration without loss of information.

The files created by this class can then be visualized as graph by a tool called "Dotty" from "Graphviz" (see Appendix; X.a.). At this point I would like to give an example of a "graph.txt" file for those who never had used this tool before:

```
digraph "security bug catcher" {
  A1 [
    height = "0.4",
    width = "0.4",
    color = "green3",
  ];
  B2 [
    height = "0.4",
    width = "0.4",
    color = "green3",
  ];
  B3 [
    height = "0.4",
    width = "0.4",
    color = "black",
  ];
  A4 [
    height = "0.4",
    width = "0.4",
    color = "black",
  ];
  A5 [
    height = "0.4",
    width = "0.4",
    color = "black",
  ];

  A1 -> B2 [
    label = "USER 1 : 331"
    color = "green3",
  ];
  B2 -> B3 [
    label = "USER 1 : 331"
    color = "black",
  ];
  B2 -> A4 [
    label = "USER 3 : 501"
    color = "black",
  ];
  B2 -> A5 [
    label = "USER 4 : 530"
    color = "black",
  ];
}
```

III.d. Statistics and error-log writer

The constructor of the '*Statistics*' class takes as arguments:

- a vector with all possible command-syntax combinations. Example:

```
[USER 1, USER 3, USER 4, USER 5, USER 6, PASS 2, PASS 3, PASS 4, PASS 5, PASS 6, REIN 3, REIN 4, REIN 5, REIN 6, ACCT 3, ACCT 4, ...]
```

- a vector containing vectors for every error id with their description. Example:

```
[[id_1, Server not Responding to current Syntax (but connection alive)], [id_3, Server not Responding to current Syntax (SERVER CRASHED)], ...]
```

- the instance of the '*Display*' class

To prevent confusion it has to be mentioned at this place, that the error with id 2 has been removed from the program (see section V.d.). That's the reason for the gap between id 1 and 3.

The '*Statistics*' class is instantiated by the '*Explorer*' and then prepares a Vector '*results*' for later storing of all errors (and also non-errors as we'll see later). The '*results*' vector is given to the '*FileLog*' constructor by creating an instance of it, so that the '*FileLog*' always possesses the newest version of Vector '*results*':

Vector results: The vector contains again vectors with information on the errors occurred.

Already at the time of its creation in the constructor of the '*Statistics*' class the entire '*results*' vector is prepared with the first two arguments showed above that are given to the constructor of '*Statistics*'. An Example of an initial '*results*' vector is given here:

```
[[USER 1, id_1, ], [USER 1, id_3, ], [USER 1, id_4, ], [USER 1, id_5, ], [USER 1, id_6, ], [USER 3, id_1, ], [USER 3, id_3, ], ...]
```

After the treatment by the '*storeError*' method that we'll see above an inner vector of the vector '*results*' could look like that:

```
[USER 6, id_4, always, NEXT ONE IS AN ERROR, Commands: [USER 1, USER 6], Server Reply: sock_close]
```

For posting an error, the '*Explorer*' uses the public '*storeError*' method of the '*Statistics*' class. Arguments appended are:

```
String syntax, String error_id, Vector pastCommands, String server_reply
```

Here the new entered error passed by the arguments is compared with the ones stored in the '*results*' vector to find out if an error occurs always on this command with this syntax, never or just at this specific moment (there may be a dependence on the precedent command sequence). This information is stored in the '*results*' vector in element at index 2. possible strings are: "" in the initial vector, "never" if only '*no_error*' errors have been set, "always" if there was always an error on this '*syntaxCombination*' and "sometimes" if there wasn't always an error on this '*syntaxCombination*'.

That's the reason why also '*no_errors*' are sent to be analysed.

The following diagram shows how the field at index 2 is modified on possible errors that are stored.

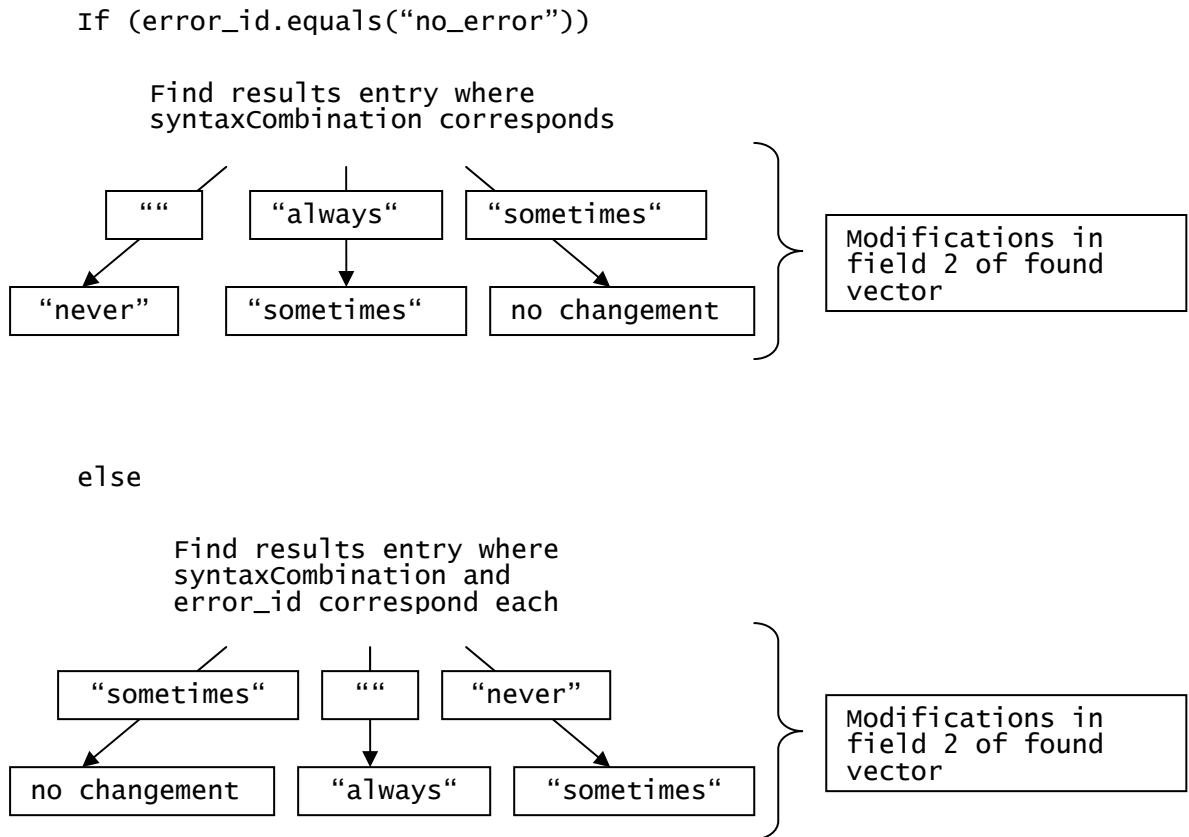


Figure 2: changing of the state ("", "never", "always", "sometimes") in the field at index 2

Afterwards there is added in case of "no_error":

```

((Vector)(results.elementAt(i))).addElement("IN THIS CASE THERE WAS NO
ERROR:");
((Vector)(results.elementAt(i))).addElement("Commands:
"+(Vector)pastCommands.clone());
((Vector)(results.elementAt(i))).addElement("Server Reply:
"+server_reply);
    
```

and in case of an error:

```

((Vector)(results.elementAt(i))).addElement("*** THERE WAS AN
ERROR:");
((Vector)(results.elementAt(i))).addElement("Commands:
"+(Vector)pastCommands.clone());
    
```

```
((Vector)(results.elementAt(i))).addElement("Server Reply: "+server_reply);
```

These three entries are appended on the inner vector of the specific command-syntax-id combination. They contain the title ("IN THIS CASE THERE WAS NO ERROR:", "***THERE WAS AN ERROR:"), the command sequence and the last server reply. In case of an error, there is also increased by one the error counter on the display:

```
frameDisplay.increase(error_id);
```

But if an error occurs always it is only counted once!

Finally, in the *'FileLog'* class all the information is taken and written into files. The files are updated every 10 seconds.

Concerning the error logs, I also changed the look of these files to make them clearer to read and more comprehensive. This modification can be read in section V.e.

III.e. State

The *'State'* class is a small one, nevertheless I'll give a brief summary of its functionality.

In the *'Explorer'* at the start of the program for every different state in the protocol there is an instance of *'State'* created.

It contains the following information:

Vector nextStates: Contains all possible transitions from this state, what means the possible commands with reply codes and next states.

Vector settings: Contains the server settings if there are some for this state (explanation see below).

Vector possibleCommands: Contains all possible commands that can be sent from this state, in form of strings.

Finally there exists in the *'State'* class a method called *'testSettings'*. This method tests if the settings defined in the "server_settings" are respected.

Example for such a file entry:

```
C
USER 1
PASS 2
---
```

This means that state C should only be able to reach after having entered USER 1 and PASS 2. Normally this is used to assure that a server accepts only to log in with the correct account information.

There has to be mentioned, that the method *'testSettings'* doesn't consider the order of the entered commands (which allows to log on for example with password as username and username as password without being detected by the program as an error). But we can assume that if a server has problems with logon security, he would also accept many other command combinations and the error would be recognized.

IV. Basic extensions and modifications of the program

IV.a. GUI for controlling

The GUI has changed its look and turned into an operational one that allows the user to set several parameters for exploration. After having modified all values, the exploration is started by clicking the “start” button. An exploration can be aborted without losing information and during exploration by clicking “visualize” the graph file is written, so that the intermediate results already can be seen in the dotted viewer.

The mouse pointer at the right hand side of the graphic part can be clicked to show the log files in a new window appearing.



Figure 3: GUI of The Security Bug Catcher

At this point I'll give short explanations on some parameters:

server name: It doesn't change the exploration but is used for creation of the folder where error logs are stored.

socket TO: The user can change the server timeout time. The value is interpreted in milliseconds and a server is declared not responding after this period of time without response. This value has to be adapted especially if the exploration is done over internet or LAN where the delay between the client (Security Bug Catcher) and the server gets bigger.

flush TO: This feature prevents the program from sending its command sequence faster than the servers time for responding and is only used if the chosen protocol has no multi-line definition (see section VI.d.). In this case the value has to be adapted to the circumstances like exploration over internet or LAN where the delay between the client (Security Bug Catcher) and the server gets bigger.

protocol: Mentions automatically all protocols implemented in the protocols folder. Don't forget to change the port number if the protocol is changed!

expl. depth / range: see section IV.c.

other checkboxes of "set exploration strategies": see section IV.d.

set the drawing modes: see section IV.g.

IV.b. Graph visualization

The goal was to develop a tool that is capable of showing the explored graph with its states, sent commands and received replies graphically. There are not many viewers available that know arranging a graph automatically taking as input its node and edge definitions.

I proposed to build this implementation based on the "Dotty" viewer developed by "Graphviz" (see section X.a.). The viewer takes a file as input that has a relatively simple syntax (DOT layout) and can take many arguments. An example for such a file is given in section III.c. where one also can find the technical aspects implemented in The Security Bug Catcher to write this files.

Figure 4 shows an example of a graph generated by The Security Bug Catcher and drawn with "Dotty".

I used different colours to visualize errors and the minimal tree. In the following I'll give a list of the different elements and their colours in the graph (colour names are the parameters used by the graph viewer; there exists a colour table on the web page indicated in the Appendix; X.a.). In brackets is given the internal error number that relates to the colour array index used in the '*DottyFileWriter*' class:

- main tree: green3 (*thisError* = 7)
- error id 1: skyblue1 (*thisError* = 1)
- error id 3: firebrick1 (*thisError* = 3)
- error id 4: darkorchid2 (*thisError* = 4)
- error id 5: pink1 (*thisError* = 5)
- error id 6: gold2 (*thisError* = 6)

The nodes are each named by their state name and an absolute state number. The edges labels contain the sent command followed by the absolute syntax string number (defined in the "syntaxStrings" file), a colon and the server reply code.

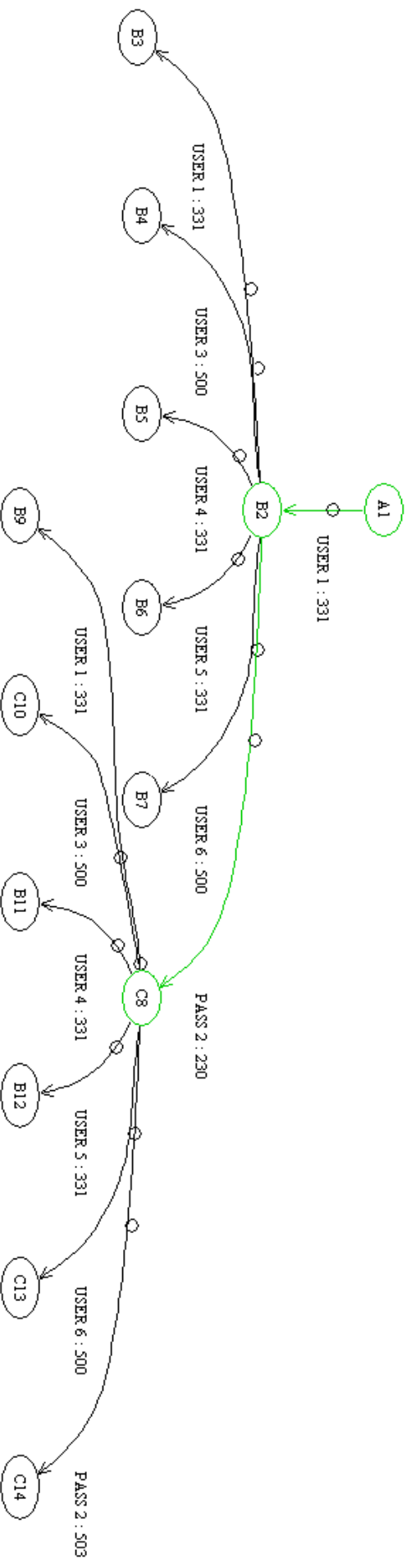


Figure 4: graph example generated by the dotty viewer from a graph file built by The Security Bug Catcher

IV.c. Strategy to limit number of server queries (minimal tree / range)

The minimal tree and range definition are in my opinion the most important strategies implemented in this second part of The Security Bug Catcher. They represent a very efficient solution to the major limitation of the initial program version that made all possible command repetitions until it reached a defined depth. As already mentioned in section II.a. the amount of explored states increased exponentially with the depth and the implementation of the entire FTP protocol would have been impossible under these conditions.

The main tree (minimal tree) I've implemented is defined by going just once through every existing state. An example for this tree of the FTP protocol is shown in Figure 5. The development of the idea to work with this tree started with the assumption that for most errors, if they occur, they can be observed very close to this minimal tree. The conclusion of the project in section IX.a. treats again this assumption and concludes if the starting idea was justified.

The Security Bug Catcher now takes as parameters a so called range which defines how many steps the exploration is continued when leaving the main tree.

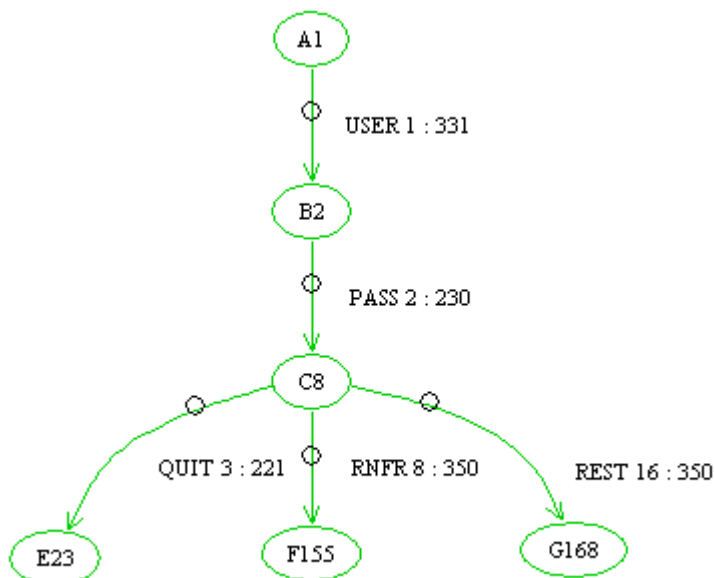


Figure 5: main tree of FTP

IV.d. Specialized exploration strategies

Apart from the range explained before I implemented other exploration strategies that are applied when errors occur. The user can choose the following strategies:

continue x steps after error: With this function, after an error (only those with id 1 or 5, for the others it makes no sense because the server closed the connection and in the case of error 6 the exploration is anyway continued), the exploration is continued x steps but for every step there are sent all possible

commands to the server because we don't know any longer in which state the server is located after an error.

continue y steps with same command: If this strategy is chosen, every time there occurs an error with id 1 or 5 (for the others this strategy makes no sense because the server closed the connection and error 6 is always continued) the last command-syntax combination is repeated y times.

reexplore same errors z times: This strategy reduces the number of explorations by exploring errors only z times that occurred from a specific state having sent a specific command-syntax combination.

The first two of these strategies have been implemented with the assumption that a server that created an error is less stable and can possibly be crashed by continuing sending commands.

The fourth choice that can be made in section "set the exploration strategies" on the GUI which is called "go back on replies defined in "goBackReplies"" is not a new one but what was called "special mode" in the initial program. The "simple mode" was removed, because it corresponds practically to the minimal tree and therefore the "special mode" was taken into the exploration strategies on the GUI. This strategy reduces the number of explorations by stopping every time an error reply appears that is set in the "goBackReplies" file of the protocol definition.

IV.e. New protocol implementation (new file structure); How to?

The initial goal was to implement the entire protocol of FTP and to test The Security Bug Catcher's compatibility with other protocols. Besides FTP I implemented the POP3 protocol which is a simpler one than FTP and works perfectly with the Security Bug Catcher.

By doing these two tasks I realized that the program structure (file structure) is not very favourable for the implementation of any other protocol without removing the existing one. I therefore decided to change the file structure and enable the adding of a new protocol just by respecting the rules on used file and folder names, put all together in the protocols folder of the program and at the next start of The Security Bug Catcher the GUI will automatically show this new protocol in the "protocols" choice field.

At this point I'd like to give a short summary, how to add a new protocol to The Security Bug Catcher. The overview is given in Figure 6. It is very important to keep all the separators (" ", "space", "---", "new line", etc.) the same as in the already existing protocols.

Important remarks:

- In the syntaxStrings file the line numbers have to be increased by one each line without gaps.
- The commandSyntax file allows to associate to one command several string groups (example: USER password,string)
- The protocol file mentions all transitions between states.

Example for "A USER 230 C":

This means that from state A with command USER and reply code 230 we arrive at state C.

- The serverSettings file sets conditions to get into a specific state (if they are not kept to, then it is defined as error with id 6). In FTP to reach state C, username and password have to be entered correctly. If our server is configured to take our string number 1 as username and string number 2 as password the corresponding serverSettings rule should be:

```
C  
USER 1  
PASS 2  
---
```

- Everything concerning the syntax strings is described in section IV.f.
- the file "regex" is used for multi-line detection which makes the program extremely performant and correct in exploration. Details on this files are given in section VI.d.

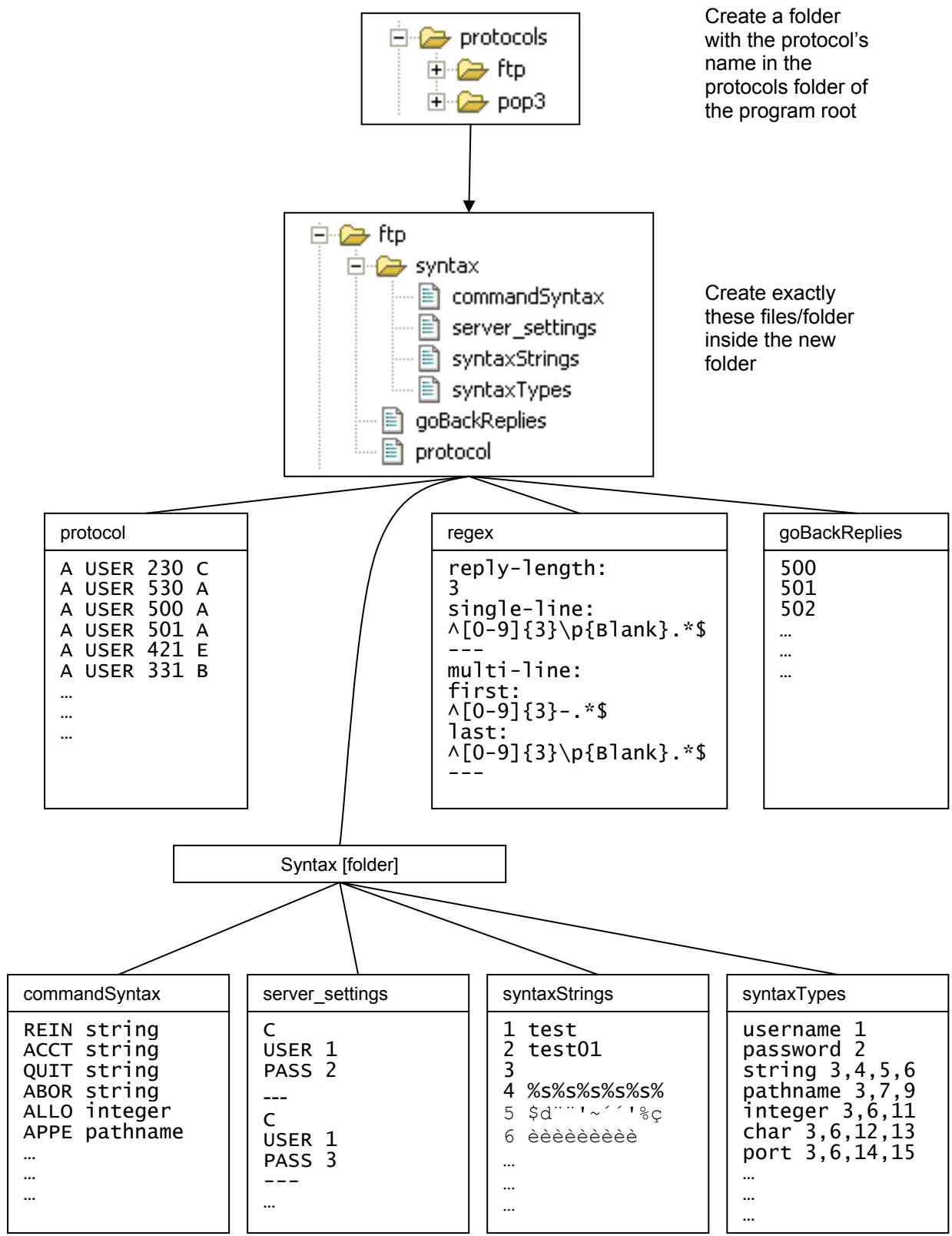


Figure 6: how to add a new protocol to The Security Bug Catcher

IV.f. Indexing and grouping of syntax strings and commands

The initial program took as input for the syntax one file for every command that looked like that (file "USER"):

```
USER test
USER test2
USER
USER anonymous
USER éLWHFALKSDJFKALSDFANMDSLKCANDSLKCAIFDS
USER %500d
USER %s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
```

Then, when the command-loop arrived at USER, it sent each line and the whole internal treatment of command with syntax was done with the entire syntax strings that sometimes contain some thousands of chars.

There have been several reasons to modify that:

- Regarding the implementation of the entire FTP protocol with about 30 commands, these definitions would have become very unwieldy.
- There would have been many repetitions of the same syntax string in different command-files.
- Replacing a string by another one in several commands and adding new strings would have become tiring.
- The error logs contained the entire strings what made them very unclear.
- Internal methods received as argument strings of sometimes thousands of chars.

Now, after several modifications the syntax strings are defined in the files as shown in section IV.f.:

"syntaxStrings" is the only file that contains the syntax strings. They are numbered increasing by one each line. The file "syntaxTypes" associates to several syntax groups (types) strings of the "syntaxStrings" file and in the "commandSyntax" file the groups are associated to the different commands.

IV.g. Drawing modes for graph visualization

When the drawing feature described in section IV.b. has been finished and some test runs had been done one had to realize that for big explorations it wasn't very nice to look at graphs with dimensions of some thousands of nodes, and sometimes the "Dotty" viewer even couldn't show the entire graphs.

At this point we decided to add a new feature (available at the bottom of the GUI, "set the drawing modes") which allows the user to draw the main tree with the branches that show errors only and leave the rest aside.

IV.h. Performance enhancement

During the coding phase of this project I frequently recognized that the program was much slower in polling the server than the server's answering. I started looking for the braking algorithms in the program and found several code segments that not only made

useless brakes but also were programmed improperly. For example timeouts were coded as infinite loops that counted till 30'000 followed by a break. These loops resulted in mistakes at error resolving during server exploration because on fast computers these timeouts became too short and therefore the program registered several server timeouts that weren't really true. Also by testing servers over LAN or internet where the delay is increased, the same effect described before came up. Now, the goal was to insert timeouts that are defined by absolute time values and to put them only in the code where they are absolutely needed, to avoid useless sleeping of the program. That involved some challenges on reply timeouts and server closing recognition which are described in section VI.b. Removing all the loops during resending of commands and in other parts of the program was done by analysing multi-line replies, which hasn't been done before and is unavoidable for performance and assuring no program errors in exploration.

Finally there have been astonishing results on performance. To document this I took the program how it was before the performance enhancement and made the same explorations later with the modified version.

An exploration with default parameter values on a Microsoft FTP server ended with following exploration times (measured by the program itself by taking the difference between end- and start- system time):

before: 120 sec.

after: 16 sec.

A reduction of exploration time of about 85% can be observed.

That shows the importance of this performance related modifications. In the case of protocols whose multi-line replies can't be described in the given "regex" file pattern the performance depends on the flush TO entered in the GUI (see section VI.d.).

V. Smaller extensions and modifications

V.a. Structured error-logs

If one wanted to make a sequence of server tests, for every exploration he had to copy the results in another folder and rename them, because The Security Bug Catcher would have overwritten the last error logs when performing a new exploration.

In view of the server analysis at the end of this project, where many servers are tested with several parameter settings, I decided to make the program ordering the error log files by the parameters given for an exploration.

In the “errorlog” folder that is located in the root of The Security Bug Catcher a folder is created with the server name entered in the GUI (microsoft ftp in the example below). In this folder a new sub-folder is created for each exploration. This folder is named with all the relevant parameters used for the exploration. Inside this folder one can find the error log files, the “summary.log” file, a copy of the “syntaxStrings” and the “graph.txt” generated by the program.

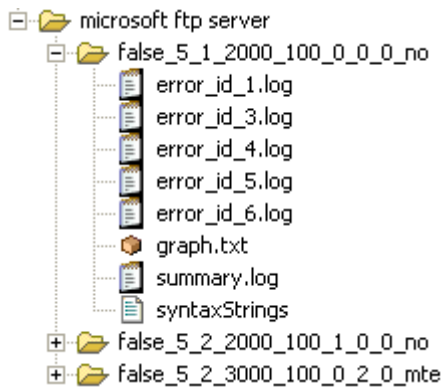


Figure 7: example of file structure created for log files

The inner folder name is created as given in Figure 8.

The “summary.log” file contains general information concerning the exploration. This is also a new feature which is described in section V.f.

Finally there is also added a copy of the “syntaxStrings” file to the exploration folder. With all these modifications the error-log of an exploration has become much more extensive and complete.

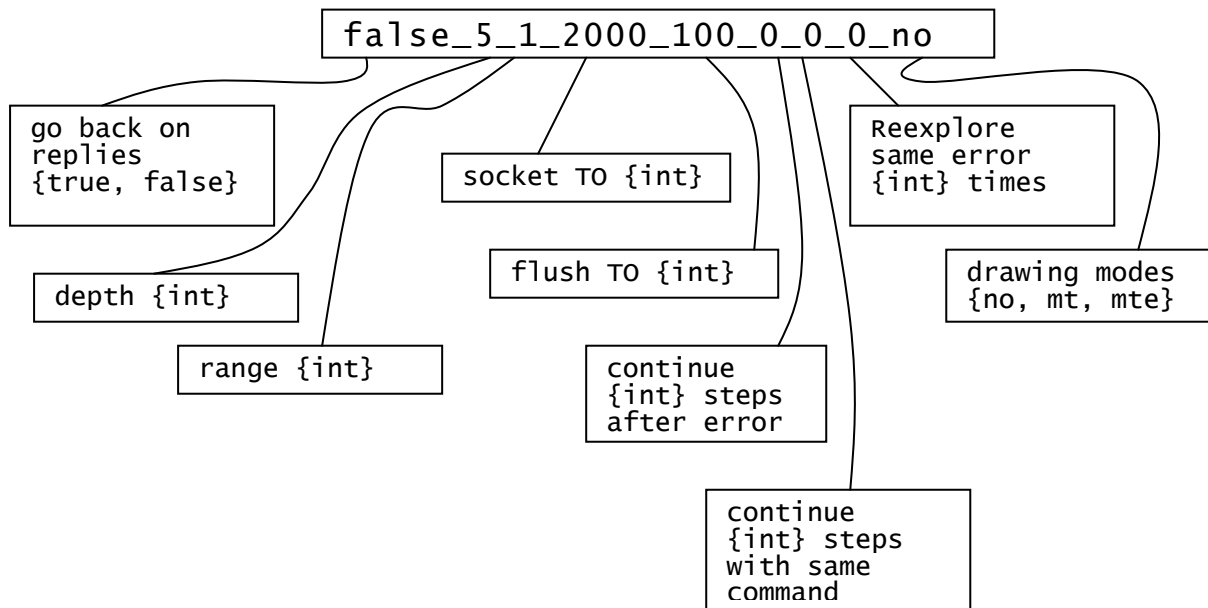


Figure 8: construction of the exploration specific folder name

V.b. Possible continuation on server crash

The program as it was designed in the first part ended improperly when the server crashed. That was not a very satisfactory solution. Therefore I first modified the code that made the program ending normally on a server crash by writing all data of the exploration till there.

We recognized also, that a server with a bug that crashes it at a certain point couldn't be explored further. That's why I introduced a popup window that informs the user about the server's crash and halts the program (without stopping!). The user now can restart the server and then click "OK" to continue exploration after crash. The crash will be shown in the tree and is stored in the error log files.

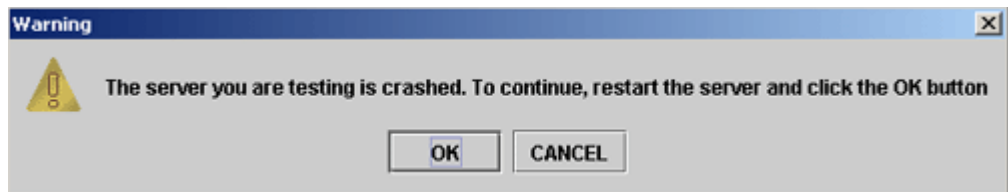


Figure 9: popup window after server crash

V.c. Visualization during exploration

Another small feature I introduced after having done the *'DottyFileWriter'* for generating the graph files, concerns the “visualize” button on the GUI.

During exploration a user can visualize the results until any point by clicking the “visualize” button. Then a dotty window opens automatically where the user can directly load the graph to look at it.

V.d. Removal of error with id 2

The abstraction between different errors made in the first part of The Security Bug Catcher was basically maintained for this second part. Except for error with id 2, which we decided to remove because its appearance is very unlikely.

Error number 2 was defined as follows:

The server doesn't respond to a command sent (timeout) but afterwards responds to another string sent to server.

Normally when a server doesn't respond (when there is a timeout), a problem occurred in the server program so that afterwards it neither won't be able to respond to another string.

V.e. Modification of error logs

It was almost at the end of this project when I decided to change the look of the error log files. As one can verify in the example below it was very hard to find particular information in the error files the way they were generated by the program in the first part. Now it is more clearly arranged and structured (see example below).

Example of an error log before modification:

```
ERROR Number 3

Frequency: sometimes
NEXT ONE IS NOT AN ERROR
-----
Commands: [USER 1, USER 3, USER 1]
-----
Server Reply: 331 Password required for test.
-----
NEXT ONE IS AN ERROR
-----
Commands: [USER 1, USER 6, USER 1]
-----
Server Reply: conn_close_before
-----
NEXT ONE IS NOT AN ERROR
-----
Commands: [USER 1, PASS 2, USER 1]
-----
Server Reply: 331 Password required for test.
```

```

-----
NEXT ONE IS NOT AN ERROR
-----
Commands: [USER 1, PASS 2, USER 1, USER 1]
-----
Server Reply: 331 Password required for test.
-----
NEXT ONE IS NOT AN ERROR
-----
Commands: [USER 1, PASS 2, USER 3, USER 1]
-----
Server Reply: 331 Password required for test.
-----
#####

```

Example of the same error log after modification:

```

COMMAND-SYNTAX-ID COMBINATION #2:
last command sent: USER 1
Frequency: sometimes

-----

IN THIS CASE THERE WAS NO ERROR:
  Commands: [USER 1, USER 1, USER 1]
  Server Reply: 331 Password required for test.
-----

*** THERE WAS AN ERROR:
  Commands: [USER 1, USER 6, USER 1]
  Server Reply: conn_close_before
-----

IN THIS CASE THERE WAS NO ERROR:
  Commands: [USER 1, PASS 2, USER 1]
  Server Reply: 331 Password required for test.
-----

IN THIS CASE THERE WAS NO ERROR:
  Commands: [USER 1, PASS 2, USER 1, USER 1]
  Server Reply: 331 Password required for test.
-----

IN THIS CASE THERE WAS NO ERROR:
  Commands: [USER 1, PASS 2, USER 3, USER 1]
  Server Reply: 331 Password required for test.
-----

#####

```

V.f. Error log summary writer

At the beginning of the server testing part I recognized the usefulness of implementing a little summary function that appends to the error files another one called "summary.log". The file gives summarising information of an exploration. It corresponds to the information that can be found on the GUI during exploration. An example of the contents of such a file is given here:

**** SUMMARY OF THIS EXPLORATION ****

Server not responding to current Syntax (but connection alive): .. 0
Server not responding to current Syntax (SERVER CRASHED): 0
Server closed connection: 0
Reply code is not valid: 26
Settings not respected: 0

Syntaxes sent to Server in depth 0 : 32
Syntaxes sent to Server in depth 1 : 37
Syntaxes sent to Server in depth 2 : 212
Syntaxes sent to Server in depth 3 : 28

total number of nodes explored: 310

VI. Some implementation challenges

VI.a. Fundamental disregards in server's protocol implementations

I was astonished about the bad implementation of the protocol some servers contain. This fact can also be observed when analysing the test series. The error that occurs most of all is error number 5 (reply code is not valid) which means that the protocol was not implemented correctly. At least, these errors normally don't mean security vulnerabilities.

But if there are too fundamental disregards or different reactions on same actions, it is nearly impossible to define rules that enable a program doing the exploration on these servers and detecting all events. Therefore sometimes the results given by The Security Bug Catcher must be interpreted by the user to see what happened.

Many servers just terminate the connection every time there is a malicious command (long string sent, etc.) which is not defined in the protocol (but can be treated by The Security Bug Catcher) and a very cheap way of going around possible security challenges.

VI.b. Timeout and server closing recognition

The counters already mentioned in IV.h. placed several times in the code blocked the program and made it very slow.

A first challenge was to ensure that all these counters could have been replaced. This solution is described in section VI.d. Another challenge described in this paragraph was to find a way of recognizing and distinguishing a timeout of the server or a server sided connection closing. There have been several thoughts behind all that:

First I found out that at socket creation with the method of the Socket class, `Socket.setSoTimeout(timeout_in_millis)`, a timeout can be added to the socket. Afterwards the blocking `in.readLine()` throws a `SocketTimeoutException` that can be caught.

Second I had to find that if the server closes during my `in.readLine()` I receive "null" as response.

Third, if the connection is closed by the server and afterwards I start my `in.readLine()` this line throws a `SocketException`.

All these characteristics lead to a clean solution for distinguishing the different cases. For completeness I'll give here the code of the reply reception part that contains all the ideas mentioned above:

```
// receiving response
try {
    inStr = in.readLine(); // blocks if connection is not closed

    if (inStr == null) { // if socket closed
        System.out.println("socket closed");
        closedWithoutTimeout = true;
        inStr = "sock_close";
    }
}
```

```

// first received line read
} catch (SocketTimeoutException e) { // after the set timeout time
//without response on the

// readLine command
System.out.println("socket exception = timeout");
inStr = "timeout";
noResponse = true;

} catch (SocketException se) { // server has been closed one
//iteration before
inStr = "conn_close_before";
stats.storeError(command + " " + syntax[k], "id_4",
executedCommands, inStr);
thisError = 4;
next_state = unknown;
goBackToPrecedentState(s, command, "conn_close_before", true, k,
thisNodeNumber);
thisError = 0;
break mainLoop;
}

```

VI.c. Graph dimensions

When I started the project, we expected drawing the tree in a java window. This is a possibility provided by a package of "Graphviz". So, the first attempt consisted in drawing the tree in a java frame. After some simulations I had to recognize that these objects created by the package (an instance for every node/edge) took immense size and after having drawn about 150 nodes, the entire memory space allocated for the JVM (about 80MB) had been used up. This clearly could not work for our project and the next approach with generating the DOT file was successful and applicable for The Security Bug Catcher.

VI.d. Treating multi-line replies

During the testing of a server over LAN, the delay led to a problem which produced mistakes in The Security Bug Catcher's error interpretations.

Up to now, before sending anything to the server, the commands of the precedent states have been resent, a response has been awaited and afterwards the in-stream buffer was flushed. Then, after having sent the command, the program waited for a response (blocking the program's continuation). The first line arriving at the buffer has been read and the first token was taken as server reply code.

This was the normal course of steps done by The Security Bug Catcher. Now, imagine a server that is responding slowly or that is distant to the client (The Security Bug Catcher) so that there is a delay of his responses. After having resent the precedent commands, the first line of response is awaited and then the buffer is flushed.

Until now, this still works very well if there is only one line of server reply. The problem was situated in multi-line replies. They haven't been treated in the program before which made it impossible to assure flawless explorations and adequate performance.

That's why I added to the protocol definitions a file called "regex" that contains information about single- and multi-line reply formats. One can define the characteristics of a single line and those of the first and the last line of a multi-line reply. As an example I'll list the "regex" files for FTP and POP3:

FTP:

```
reply-length:
3
single-line:
^[0-9]{3}\p{Blank}.*$
---
multi-line:
first:
^[0-9]{3}-.*$
last:
^[0-9]{3}\p{Blank}.*$
---
```

POP3:

```
reply-length:
4
single-line:
^\(+OK )|(-ERR ).*$
---
multi-line:
first:
no
last:
no
---
```

The line after "reply-length:" contains the number of digits of the reply code. After "single-line:" a regular expression that describes the general architecture of a single line has to be indicated and after "multi-line first/last" the pattern of the first / last line of a multi-line reply has to be mentioned. If a protocol doesn't make this difference between single and multi-line reply or it can't be implemented with this pattern, there can just be given a single line regular expression and a reply-length for parsing the reply and for the multi-line "no" is put in. That means that the security bug catcher now works with timeouts as before and is not as performant as with enabled multi-line feature.

With this information the program always waits until the end of the reply by knowing if a single- or multi-line reply was sent. That's how all the waits in the code have been removed and therefore performance has been increased incredibly.

A big advantage is also that there is no dependence whether there is a delay between server and program because we always exactly know when all information is received and we can go on.

For a fast server running on the same computer like The Security Bug Catcher values around 100-200ms as flush TO may be chosen.

VII. FTP Server examination reports

setting the context

FTP server tests have been done on Windows (WindowsXP) and Linux (Debian) stations. The operating system is indicated in brackets behind the server name.

I tested every server with several different strategies, ranges, timeouts, etc. to cover the biggest range of possibilities for finding errors.

For this exploration run the following syntax strings have been used:

```
1 test
2 test01
3
4 %s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
  s%s%s%s%s
5 $d`~'%'ç*"(/)+\<>-ö{}}]éàf!è`?__§°10
6 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  ... (4200 chars)
7 test/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  ... (4200 chars)
8 /%s%s%s%s%s%s%s/s/%s%s%s%s%s%s%s/s/s
9 /testfolder/
10 /testfolder/test.file
11 1234944982918021231234123444596970
12 %
13 $
14 300,300,-200,12345123566123,0,-123
15 .....
16 1000
17 .
18 asdf aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaa... (4200 chars)
19 asd fsd kl we lg asdf ou asduo iuz zt gf sdf s re de fg hg v b j
  jbh gf hgf s s rew s fdsg fc sdfg fb cvb dfg sdf we r fdf gdf g
  bdsf sdf sdf gd bfc vxc vxcv xcv xcv sd fas fdwd ff gfg gfn df sd
  ... (150 short words)
20 -1234944982918021231234123444596970
21 12.2.3.4.54.6.7.5.4.3.2.3.4.6.4.4.5.3.2.3.4.5.6.3.4.2.3.3.44.3.2.32
  .3.4.5.4.3.23.34.5.6.4.23.34.4.4.5.3.3.4.5.4.3.4.4.5.5.4.3.4.5.34
  .5.4.5.4.34.5.34.35.35.3.2.23.4.4.3.3.2.3.4.3.4.3.2.34.2.3.4
22 //////////////////////////////////////
```

Some server specific parameters as username, password and test folders have been adapted to test runs.

VII.a. BisonFTP Server V4R1 (Windows)

This server has a very special behaviour on the port command with almost every possible string. The consequences of a PORT command are that the server doesn't

respond, but (as one can see on the server interface) he keeps all that is sent to him in the following and concatenates it by raising exceptions.

A repetition of the command port (without argument) therefore looks like that in the server interface:

```
127.0.0.1-PORT |PORT |PORT |PORT |PORT |PORT |PORT
Exception Raised: " is not a valid integer value
127.0.0.1-PORT |PORT |PORT |PORT |PORT |PORT |PORT |PORT
Exception Raised: " is not a valid integer value
127.0.0.1-PORT |PORT |PORT |PORT |PORT |PORT |PORT |PORT |PORT |PORT
Exception Raised: " is not a valid integer value
```

Figure 10: snippet of server logs

When I remarked that, I started an exploration with the new implemented strategy to produce 10 repetitions of the same command in case of an error. Interesting things occurred. After a certain number of commands the server starts making address access violations and one can't connect any longer from the same IP.

Server reply: "421 too many connections from your IP address"

So I tested continuing from another IP. The result was, that after about three commands the server didn't respond any longer (no commands worked, even a "quit" was impossible) and also on the server interface, connections haven't been shown (it was almost frozen; only exception logs have been shown any longer).

At this point the server log looked like shown in Figure 11.

The graphical interface of the server was still accessible; therefore I stopped the service and restarted it. That worked very well but connecting to the server was still impossible with the same error mentioned above.

Interpretation:

- Port commands invoke problems that lead to address access violations
- An open session can't be closed after having sent a malicious PORT command and even the server is not capable of closing it after the timeout (that's why he doesn't accept any new users). Probably he occupies many sockets with open connections and is getting blocked when their number reaches a certain value.

```

Exception Raised: Invalid pointer operation
Exception Raised: A component named frmConnection already exists
Exception Raised: " is not a valid integer value
Exception Raised: A component named frmConnection already exists
Exception Raised: " is not a valid integer value
Exception Raised: Access violation at address 0044E8E1 in module 'Bisonftp.exe'. Read of address 000002CC
Exception Raised: A component named frmConnection already exists
Exception Raised: " is not a valid integer value
Exception Raised: A component named frmConnection already exists
Exception Raised: " is not a valid integer value
Exception Raised: Access violation at address 0044E8E1 in module 'Bisonftp.exe'. Read of address 000002CC
Exception Raised: Access violation at address 0044E8E1 in module 'Bisonftp.exe'. Read of address 000002CC
Exception Raised: A component named frmConnection already exists
Exception Raised: " is not a valid integer value
Exception Raised: " is not a valid integer value
Exception Raised: " is not a valid integer value
Exception Raised: " is not a valid integer value
Exception Raised: Access violation at address 0044E8E1 in module 'Bisonftp.exe'. Read of address 000002CC
Exception Raised: " is not a valid integer value
Exception Raised: " is not a valid integer value
Exception Raised: " is not a valid integer value
Exception Raised: Access violation at address 0044E8E1 in module 'Bisonftp.exe'. Read of address 000002CC
Exception Raised: A component named frmConnection already exists
Exception Raised: A component named frmConnection already exists
Exception Raised: A component named frmConnection already exists
Exception Raised: A component named frmConnection already exists
Exception Raised: Access violation at address 0044E8E1 in module 'Bisonftp.exe'. Read of address 000002CC
Exception Raised: A component named frmConnection already exists
Exception Raised: Access violation at address 0044E8E1 in module 'Bisonftp.exe'. Read of address 000002CC
Exception Raised: Access violation at address 0044E8E1 in module 'Bisonftp.exe'. Read of address 000002CC
Exception Raised: Access violation at address 0044E8E1 in module 'Bisonftp.exe'. Read of address 000002CC

```

Figure 11: snippet of server logs

VII.b.oftpd 0.3.6-5.1 (Linux)

This server makes a denial of service every time after sending the “PORT” command followed by 300,300,-200,12345123566123,0,-123. Afterward it must be restarted manually.

Besides of that there are just fault of protocol implementation and no timeouts.

VII.c. Broker 6.1.0.0 (Windows)

The server responded very slowly and there were some timeouts, but it never crashed.

VII.d.Serv-U 5.0 (Windows)

There have been no timeouts with range equal 1.

Increased range brought many timeouts and protocol implementation faults. This is rather unusual that increasing range also brings much more errors.

VII.e. Ability FTP Server (Windows)

This is one of the few servers that never had a server timeout or closed a connection during exploration. It only presented some protocol implementation faults. Some answers are arriving very late, that's why I had to use big timeout times (like 5secs).

VII.f. wu-ftp 2.6.2 (Linux)

There are only faults in protocol implementations. In the whole test series no timeout was produced and the server never closed the connection.

VII.g. Pure-FTPd 1.0.17 (Linux)

Applying the strategy to go on one step after a command produces in this case many timeouts.

VII.h. vsFTPD 1.2.1 (Linux)

For this server a big exploration has been done (more than 40'000 pollings). "Not Responding" errors only have been produced on range two exploration. Else there were very few errors.

VII.i. Microsoft FTP Server (Windows)

With range 1 and without any strategies there are no server timeouts and no intentional connection closings. By applying strategies some intentional connection closings are forced and in range 2 there are many timeouts.

For this server there has been made an exploration containing 170'000 server pollings. This was possible, because of the fast responding of the Microsoft FTP Server and the new implementations for high performance in The Security Bug Catcher.

VII.j. WS_FTP Server 4.02 (Windows)

There are no server timeouts and the server almost never closes a connection intentionally.

VII.k. RaidenFTPD 2.4 (Windows)

Raiden FTPD systematically produces timeouts or intentionally closes the connection.

VII.I. Black Moon FTP Server 2.8 (Windows)

This server never had a timeout during all the tests. There have only been protocol implementation faults.

VII.m. ArGoSoft FTP Server 1.4.1.3 (Windows)

There are no timeouts when exploring this server but at each long string he writes in its interface:

“Error: Buffer is full. Will disconnect”

That’s why we get many server connection closings in each exploration.

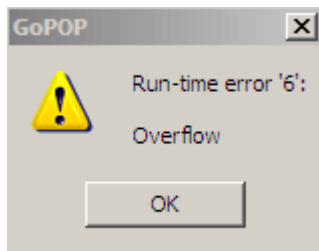
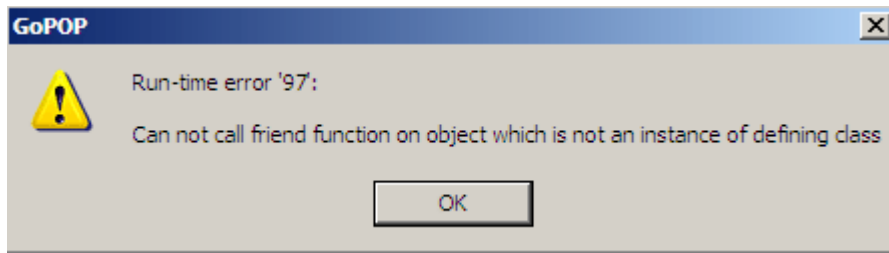


Figure 12: popups of GoPop when sending long strings

Figure 13 shows the tree of the exploration until the first server crash of GoPop.

VIII.b. Inframail 6.17 (Windows)

There are a few timeouts in every exploration.

VIII.c. teapop 0.3.5 (Linux)

Teapop doesn't produce many errors but there are a few in every domain.

VIII.d. Solid POP3 Server (Linux)

In the default exploration (with default parameters) the server stopped several times and had to be restarted manually. But by repeating the same scenario of command repetitions afterwards by hand (from console) there wasn't the same result (the server didn't stop). And also by repeating the same exploration the server stops are at different places.

There may be a problem of memory allocation during time which produces this denial of service after some malicious commands.

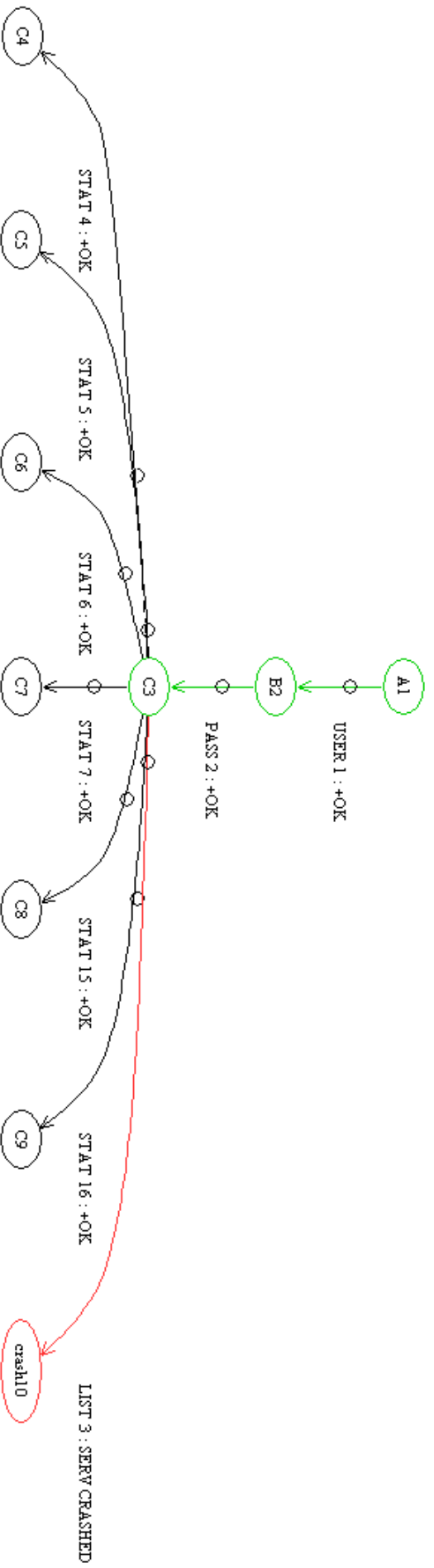


Figure 13: tree of exploration of GoPop until the first crash

VIII.e. popa3d (Linux)

This server often sends null after opening which is confusing The Security Bug Catcher who interprets this as server crash. But the server is not crashing.

VIII.f. mailutils-pop3d (Linux)

Like the Solid POP3 server this one also stops and has to be restarted manually. The Security Bug Catcher receives in this case "null" on connection opening. This kind of server closings are not counted as server crashed (the user is only informed by a popup window) by The Security Bug Catcher because the closings appear not where normally a crashed server stops responding.

VIII.g. TmPopAdmin 2.13 (Windows)

This server commits absolutely no timeouts or connection closings and very few implementation faults.

IX. Conclusion

IX.a. Conclusion of the server test series

The test series contained many different server implementations of different operating systems. Overall, there have been running tests during about 30 hours. More than 300'000 server queries have been sent, spread over about 20 different server implementations.

Summary of the most important vulnerabilities that have been found:

- Bison FTP → very special behaviour (see VII.a.)
- oftpd → denial of service after specific commands
- GoPop
- Solid POP3 Server → denial of service at undefined moments
- mailutils-pop3d

None of the server crashes occurred only in range 2 but always already in range 1. With this idea we started to implement the minimal tree and the synthesis of this test series are proving that we probably were right. And often by searching not far from the minimal tree (in range 1) we still find most of the less dangerous server errors. But there have been some servers that produced for example timeouts only for range 2 explorations.

Secondly the idea of going on after an error (because of possible instability) by implementing strategies has been paid. The very particular error in the Bison FTP server could only have been found by applying these strategies.

It is also clear that a big part of success or not by finding errors with The Security Bug Catcher depends on the choice of the syntax strings delivered to the program. In this project, the testing period was very short and with more specialization on the syntax sent to the server there could probably again be found some vulnerabilities.

One could imagine to make explorations with only one simple string which is normally accepted for every command but to put in a big amount of different port numbers (maybe even generated by another program) to attack very specific vulnerabilities (in this case commands that take port numbers).

In POP3 there have been observed much less protocol implementation faults than in FTP which is relatively logic because of the increased complexity of the FTP protocol compared to POP3.

Summary of the explorations for the five main server vulnerabilities found:

- Range: All crashes can be found with range = 1 (→ big performance enhancement compared to the exploration of all possible combinations)
- Special strategies for continuation on error: vulnerability in BisonFTP can only be found with the two strategies that continue exploring after an error.
- Average number of nodes explored until crash detection: ~120
- Average depth where crash occurred: depth = 3 (→ after login)
- Special strategy for stopping on same error repetition: Is not necessarily used in these cases, but reduces time in explorations of range 2 and more.

IX.b. Conclusion of the entire project

The result of the project is in my eyes very satisfying. All the goals mentioned at the beginning have been fulfilled and there have been added many new features to The Security Bug Catcher. In addition many other problems have been faced and have continuously been implemented.

In my opinion The Security Bug Catcher is now widely developed, supports very fast, efficient server polling with a big choice of strategy combinations and complete error logs with its representation in a tree.

For me personally, this project was very interesting and enabled gaining experience in the domains of server protocols, server implementations, security vulnerabilities and Java programming.

Finally I also want to thank Philippe Oechslin for his effort and the very pleasant collaboration during the whole project.

X. Appendix

X.a. Dotty / Graphviz

This open source graph drawing tool is described and available at the following link:
<http://www.research.att.com/sw/tools/graphviz/>
In general it interprets graphs in the DOT layout.

X.b. RFC's

All existing RFC's can be found under:
<http://www.ietf.org/rfc.html>

RFC numbers:
FTP: 959
POP3: 1081

X.c. Code of the entire 'explore' method

Starting on next page.

```

// main method that recursively sends commands to server and makes the analysis (this is the central engine of the program)
public void explore(State s) {

    int thisNodeNumber = nodeCount; // keep the nodecount to link new nodes later
    System.out.println("\nnew iteration of explore\n");
    thisError = 0;
    String stateName = s.getName();
    String command;

    Vector stateCommands = s.possibleCommands; //duplicates the commands initial vector

    boolean goNotBackBecauseReply = true;

    if (!stopIteration(stateName)) {

        mainLoop:
        for (int i = 0; i < stateCommands.size(); i++) { // for each command (i)
            command = (String) stateCommands.elementAt(i);
            int[] syntax = (int[]) (idsOfCommands.get(command)); // contains array of syntax ids for this command

            //System.out.println("exploring from state: " +name+ " command: " +command+ " depth=" +depth);

            syntaxIteration:
            for (int k = 0; k < syntax.length; k++) { // for each syntax (k) of the current command
                nodeCount++;

                if (endExploration) {
                    System.out.println("endExploration, break loop");
                    break mainLoop; // in case of crash
                }

                // decision for reexplore strategy: if the next state should not be reexplored, one iteration is jumped
                if (errorReexplore && ((int[][])(reexploreArchive.get(stateName)))[i][((int[]) (idsOfCommands.get(command)))[k]] >
errorReexploreValue) {

                    System.out.println("-> don't reexplore: jump back on k = " +k+ ", i = " +i);
                    continue syntaxIteration;
                }

                if (depth == 0) { // we are in depth 0 -> line grows of one unit
                    line_display.go(step);
                }

                String inStr = "xxx";
                thisError = 0;
            }
        }
    }
}

```

```

executedCommands.addElement(command + " " + syntax[k]);

//      update parameters of the display
line_display.updateNb(depth);

System.out.println(
    "exploring from state: "
    + stateName
    + ", depth="
    + depth
    + " command: "
    + command
    + ", syntax: "
    + syntax[k]);

emptyBuffer(flushTimeout);
String reply code = "";
try { // send packet to server

    String line;

    out.println(command + " " + syntaxArray[syntax[k]]);

    out.flush();

    boolean noResponse = false;
    boolean closedWithoutTimeout = false;

    // receiving response
    try {
        inStr = in.readLine(); // blocks if connection is not closed

        if (inStr == null) { // if socket closed
            System.out.println("socket closed");
            closedWithoutTimeout = true;
            inStr = "sock_close";
        }
        // if first received line read
        System.out.println("GOT MESSAGE: " + inStr);

    } catch (SocketTimeoutException e) { // after the set timeout time without response on the
        // readLine command

```

```

        System.out.println("socket exception = timeout");
        inStr = "timeout";
        noResponse = true;
    } catch (SocketException se) { // server has been closed one iteration before
        inStr = "conn_close_before";

        stats.storeError(command + " " + syntax[k], "id_4", executedCommands, inStr);
        thisError = 4;
        next state = unknown;
        goBackToPrecedentState(s, command, "conn_close_before", true, k, thisNodeNumber);
        thisError = 0;
        break mainLoop;
    }

    if (noResponse || closedWithoutTimeout) { // server produced a timeout

        if (noResponse) {
            if (checkConnectingPossible()) {
                stats.storeError(command + " " + syntax[k], "id_1", executedCommands, inStr);
                if (errorReexplore)
                    ((int[][]) (reexploreArchive.get(stateName)))[i][k]++;
                thisError = 1;
            } else {
                stats.storeError(command + " " + syntax[k], "id_3", executedCommands, inStr);
                if (errorReexplore)
                    ((int[][]) (reexploreArchive.get(stateName)))[i][k]++;
                thisError = 3;
                System.out.println("SERVER CRASHED");
                boolean answer = showCrashDialog(); // to restart server and continue exploration in
                if (answer) {
                    goBackToPrecedentState(s, command, "SERV CRASHED", true, k, thisNodeNumber);
                    continue;
                } else {
                    endExploration = true;
                    break mainLoop;
                }
            }
        } //else
        System.out.println("go back 256");
    }

```

case of crash

```

next state = unknown;
goBackToPrecedentState(s, command, "timeout", true, k, thisNodeNumber);
continue;
} else
if (closedWithoutTimeout) {

    if (checkConnectingPossible()) {
        stats.storeError(command + " " + syntax[k], "id_4", executedCommands, inStr);
        if (errorReexplore)
            ((int[][])(reexploreArchive.get(stateName)))[i][k]++;
        thisError = 4;
        System.out.println("go back 267");
        next state = unknown;
        goBackToPrecedentState(s, command, "close_conn", true, k, thisNodeNumber);
        continue;
    } else {
        stats.storeError(command + " " + syntax[k], "id_3", executedCommands, inStr);
        if (errorReexplore)
            ((int[][])(reexploreArchive.get(stateName)))[i][k]++;
        thisError = 3;
        System.out.println("SERVER CRASHED");
        boolean answer = showCrashDialog(); // to restart server and continue

        if (answer) {
            goBackToPrecedentState(s, command, "SERV CRASHED", true, k,

                continue;
            } else {
                endExploration = true;
                break mainLoop;
            }
        }
    }
} //if

} catch (IOException e) {
    System.out.println("error in in.readLine()");
    e.printStackTrace();
} catch (NullPointerException ne) {
    System.out.println("SERVER NOT STARTED");
    serverNotStartedDialog();
    endExploration = true;
    break mainLoop;
}

```

exploration in case of crash

thisNodeNumber);

```

reply_code = getErrorReply(inStr);

//      Check if the reply code is an error reply from the error_replies file
if (goBackRepliesBool) {
    for (int l = 0; l < goBackReplies.size(); l++) {

        if (((String) goBackReplies.elementAt(l)).equals(reply_code)) {
            System.out.println(goBackReplies.elementAt(l)+ ":"+reply_code);
            goNotBackBecauseReply = false;
        }
    } //for
} //if

//      ----- Beginning point of what to do after this state...

System.out.println("REPLY CODE: " + reply_code);
next state = findNextState(s, command, reply_code);
System.out.println("AFTER FINDING NEXT STATE");

if (goNotBackBecauseReply) { //if reply is not one from the goBackReplies file

    if (next_state.getName().equals("unknown")) { //next state does not exist

        stats.storeError(command+ " " +syntax[k], "id_5", executedCommands, inStr);
        if (errorReexplore)
        ((int[][])(reexploreArchive.get(stateName)))[i][((int[]) (idsOfCommands.get(command)))[k]]++;
        //System.out.println("***** Reexplore TEST by stats: state = " +stateName+ " , i = " +i+ " ,
syntax = " +((int[]) (idsOfCommands.get(command)))[k]);
        thisError = 5;

        //System.out.println("state does not exist");

        goBackToPrecedentState(s, command, reply_code, true, k, thisNodeNumber);

    } //if

    else { //next state exists

        settings test = next state.testSettings(executedCommands);
        if (!settings_test) {
            stats.storeError(
                command+ " " +syntax[k],
                "id_6",

```

```

        executedCommands,
        "");
        if (errorReexplore) ((int[][])(reexploreArchive.get(stateName)))[i][k]++;
        thisError = 6;
    }
    // no error is used to find out in statistics if an error occurred every time in this situation
    stats.storeError(command+ " " +syntax[k], "no error", executedCommands, inStr);
    //System.out.println("go to next state (complete, special)");
    goToNextState(s, next_state, command, reply_code, k, thisNodeNumber);

    } //else
} //if current_error_reply

else { //if reply is one from the goBackReplies file

    goNotBackBecauseReply = true; //set current_error_reply to true for the next states...
    System.out.println("go back 350");
    goBackToPrecedentState(s, command, reply_code, true, k, thisNodeNumber);

} //else

} //for k (commands)
} //for i (syntaxes)

thisError = 0;
System.out.println("##### End of exploration of an intermediate node");
System.out.println("go back 363");

if (!endExploration) goBackToPrecedentState(s, "", "", false, 0, thisNodeNumber);
else {
    if (!crashNodePainted) {
        //int synt = ((int[]) (idsOfCommands.get(command)))[0];
        drawGraph(s.getName(), "crash", (String) executedCommands.elementAt(executedCommands.size()-1), -1, "SERV CRASHED",
3, thisNodeNumber);
        crashNodePainted = true;
    }
}

} // if (!stopIteration)

else {

    System.out.println("else depth < depth_stop && !name.equals(exitStateName) && settings_test");

    if (!settings_test) {

```

```
        settings_test = true;
        System.out.println("-> because of settings_test");
    } else {
        if (!(depth < depth_stop))
            System.out.println("-> because of depth");
        else {
            if (!(range < range_stop))
                System.out.println("-> because of range");
        }
    }

    goBackToPrecedentState(s, "", "", false, 0, thisNodeNumber);
} //else
} //end of explore
```