



Flinder

We make your software secure

Flinder test methodology overview

Gergely Tóth
Zoltán Hornák

April 10, 2006

CONTENTS

1. Abstract.....	3
2. Introduction.....	4
3. The Flinder approach.....	5
3.1 Black-box testing.....	6
3.2 White-box testing.....	8
4. Flinder at work.....	10
4.1 The MFDL.....	11
4.2 Finding the buffer overflow.....	12
5. Conclusion.....	15
6. References.....	15

1. ABSTRACT

Exploitable security vulnerabilities cause huge damages year by year. The problem is that creating secure applications is expensive, since current testing-based (cheaper) approaches are not effective enough to substantially improve security, so the only remaining proven solution is formal verification which is far too expensive for general use. The main goal of our test-based approach – called *Flinder* – is to provide a framework for software developers, which could easily be integrated in the current software development lifecycle and could perform semi-automated security testing. The main advantage of Flinder is that instead of formal specification and verification or extensive testing it relies basically only on syntactically and semantically correct test inputs, where the appropriate answer of the system was tested beforehand, and the message format description (MFDL) of these test messages to let manipulation of test vectors in order to expose the security-relevant programming bugs. In practice, Flinder operates as a man-in-the-middle between an existing test generator and the target of evaluation (ToE), and injects special modifications to the generated test vectors and observes the response of the ToE. The test algorithm then iteratively generates the next test vectors based on the operation of the ToE in order to locate where the reaction of the ToE changes, because most security-relevant bugs can be found in such circumstances. We tested this approach in a pilot with existing software by re-discovering known bugs.

Acknowledgement

The work presented in this paper has partially been funded by the Operative Program for Economic Competitiveness of the Ministry of Economy and Transport and the Ministry of Education under contract number GVOP - 3.3.1 - 2004 - 04 - 0094/3.0.

2. INTRODUCTION

Programming bugs can be grouped in three self-containing sets: (1) *programming bugs*, which affect functionality in general, (2) *security-relevant programming bugs* that can, but do not necessarily degrade the security properties of the system (we may also call them “dangerous, but not necessarily exploitable” bugs and (3) *exploitable security vulnerabilities*, where the programming bug can be exploited and this can cause a security breach. In practical terms we may call a system secure if there are no exploitable security flaws in it. However, to formally prove that the system is secure, we would have to prove that it fully complies with its specifications, that is no programming bugs are in the system. While it is hard to address the first goal directly by cheap but satisfactory testing or verification methods, on the other hand the formal verification is very expensive and unnecessarily strict, since the security goals require only that there are no exploitable flaws, and the functional correctness is not a necessity. However, in case of most typical security-relevant programming bugs, effective testing methods can be applied to reliably discover sources of these bugs. By discovering and eliminating these dangerous (but not necessarily exploitable) bugs, we can avoid the great majority of typical exploitable vulnerabilities. So the main goal of Flinder is to *detect security-relevant programming bugs* during the software development process before they turn into exploitable vulnerabilities.

In traditional secure software engineering the emphasis was on formal methods (which could prove the correctness of the applied techniques) and on extensive testing. Flinder’s aim is to provide additional help in testing by utilizing a new approach for test vector generation. In our concept the ToE is communicating with an input generator via messages. The idea is that Flinder modifies these messages in a man-in-the-middle way. Naturally, this communication can be network-based, but a simple application processing files can also be handled this way.

In order to be able to modify the input messages Flinder needs to know the format descriptions of the different messages. Based on the message format descriptions Flinder transfers each message into a general internal format (MSDL). Test specific modifications (so-called *TestLogic*) will work on this internal representation. It is also possible that one test step consists of not just one request-response message exchange, but a series of messages (i.e. execution of a protocol) is needed to drive the ToE into the targeted state, and Flinder has to modify the content of a message only then. For testing such protocols, format description of each protocol message and the protocol’s state chart have to be given. For this reason Flinder maintains a Protocol Statechart (based on a UML state machine), which can describe the series of messages between the test generator and the ToE.

So Flinder can understand protocol steps and modify messages between the input generator and the ToE, aiming to reveal the security-relevant programming bugs. Generic testing algorithms are then used, that can work on the internal representation of parsed messages.

For making testing more efficient, Flinder is capable of looking for different bugs *concurrently* (e.g. by testing different buffers simultaneously). Furthermore, by taking the responses of the ToE into account, Flinder can employ *reactive* testing to better identify potential security bugs.

Based on the availability of the source code Flinder can be used in black-box or white-box scenarios:

- In the *black-box* mode the ToE is evaluated in its executable form and Flinder supplies the input directly to it and draws conclusions based on successful or abnormal reaction (e.g. OS level signals). This approach could be favoured if an independent security audit is required, and the source code is not available (e.g. 'white-hat hacking').
- *White-box* testing could be applied if the source code is available. This way Flinder could inject the modified test vectors into the tested functions directly, this way it could achieve a much bigger coverage and Flinder could be involved in the internal (source code level) testing of a product.

Later on we will present this approach for security testing and illustrate its capabilities on a simple example – finding a buffer overflow in an e-mail client.

3. THE FLINDER APPROACH

The main goal of Flinder is to detect security-relevant programming bugs based on fault injecting test vector generation. For this purpose several products and algorithms are already available, however the novelty in the Flinder approach lies in the following advantages:

- *Testing based on data descriptors*: besides input vectors that describe the correct functioning of the system in some sense, Flinder needs only the format descriptors (MFDLs¹), which describe the structure of the input, generated by an existing test application, the so-called Input generator.

Compared to other testing methods, Flinder does not need the specification of correct behaviour to discover unintended strange operation. It can create test vectors for detecting the security bugs based on valid input and the information incorporated in the message format descriptors.

- *Generic testing algorithms in a plug-in architecture*: further advantage of Flinder is that once a test message is available in a generic internal structure based on the MFDL descriptor, generic test algorithms can be used to create test vectors aiming to reveal the security-relevant programming bugs. These generic algorithms can therefore operate on various kinds and types of input data (e.g. on an XML document or an encoded SSL packet).

It is important to emphasize that such generic testing algorithms can be constructed for several types of security vulnerabilities, for example:

- *buffer-overflow type bugs* – by enlarging the size of variable-length buffers possible overflows could be triggered,
- *integer-overflow type bugs* – by trying out special numbers to cause signedness, widthness bugs and arithmetic overflow, and
- *encoding bugs* – by changing the encoding property of a field or by customly serializing different parts disregarding the encoding rules.

¹ Section 4.1 shows a sample MFDL used by Flinder to detect a buffer overflow type vulnerability in Pine 4.56.

- *Protocol Statecharts*: in order to test complex protocols with different states Flinder maintains a Protocol Statechart. This statechart is based on the UML state machine model [UML], where the transitions are either messages exchanged between the Input generator and the ToE or signals indicating some abnormal event (e.g. unexpected termination or a timeout). With the help of the Protocol Statechart Flinder can identify message types and corresponding MFDLs.

These core concepts can eventually be used for both black-box and white-box testing, which will be introduced in the following sections.

3.1 Black-box testing

When Flinder operates in black-box mode, the way of testing is the following. Flinder is inserted in a man-in-the-middle way between the *Input generator* (which generates syntactically and semantically correct test vectors for the ToE) and the *ToE*. The goal is that all communication should flow through Flinder in both directions. In this scenario the testing sequence (illustrated on Figure 1) is the following:

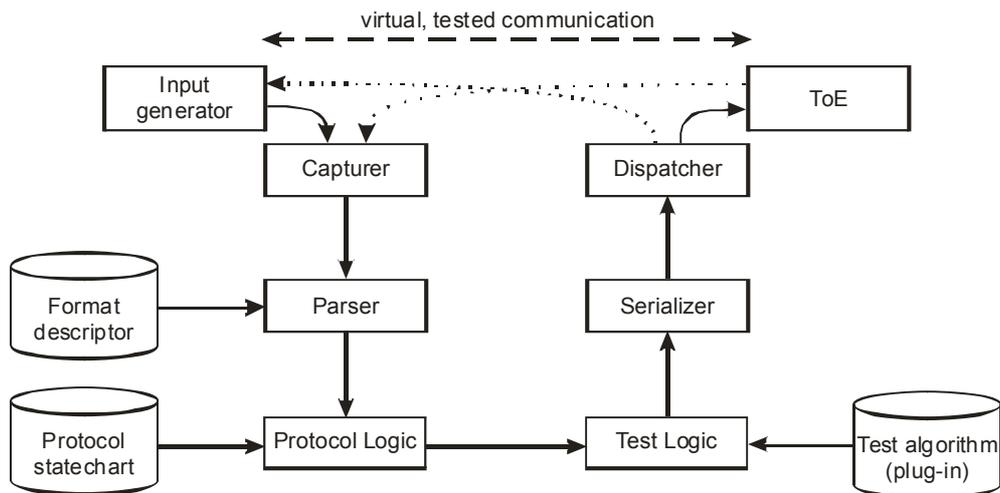


Figure 1 – Black-box testing with Flinder

1. The Input generator or the ToE sends a message to the other party.
2. The original binary message is caught by the *Capturer*. The main task of the *Capturer* is to redirect the binary message and insert it into Flinder's message processing pipe. Besides the raw data additional information can accompany the payload, such as the timestamp or other environmental parameters. All these pieces of information are stored in a BIME (Binary Message Envelope).

3. The incoming BIME is then processed by the *Parser*. The main task of the Parser is to convert the binary message into the internal structure (called MSDL²) based on the message descriptor (MFDL).

The main characteristics of MFDL are:

- It resembles most the XML Schema [SCHEMA] and is capable of describing data structures made up of primitive types (e.g. integers or strings) and compound types (e.g. sequences, lists or choices). It is possible to convert a wide variety of existing data descriptions to MFDL (e.g. ASN.1 or XML Schema).
 - Parsing is done by two entities: a generic entity manages the abstract data structure (i.e. it is responsible for creating the parse tree), while *decoders* are responsible for reading the binary stream and convert the read information into the generic internal structure called MSDL. Since there are different encoding types, MFDL was designed to express almost all possible data formats. Thus Flinder now supports generic binary encoding, like little or big-endian for integers, XML structures, the Distinguished Encoding Rules (DER) for ASN.1 structures (used for example for X.509 certificates) or separator based encoding for text-driven data (usable for example for parsing HTTP).
 - Finally, MFDL provides facilities to specify *actions* to be carried out during parsing, e.g. specifying the size of a field, based on data already parsed in another field.
4. After the message has been parsed in and the MSDL tree structure is created, the *Protocol Logic* is responsible for updating the Protocol Statechart according to the type and content of the received message. The statechart can be used to detect anomalies in the message flow and to maintain the protocol state.
 5. The main task of Flinder is done in the next step by the *Test Logic*. This module is the host of the plug-in architecture that incorporates the different generic testing algorithms, whose task is to modify the MSDL so that the respective security-relevant programming bugs should be covered. These algorithms operate on the internal data structure (MSDL). A specific example of a buffer-overflow testing algorithm will be introduced in Section 4.2.
 6. The *Serializer* module transforms the modified MSDL into the binary format and creates a BIME (similarly to the operation of the Parser).

Just like during parsing, different *encoders* are used during serialization in order to create the different binary representations. Furthermore, *actions* can also be used to enable execution of operations during the serialization process (e.g. recalculating a digital signature on a message or adjusting the size of a field after modification by the generic test algorithm).

7. Finally the *Dispatcher* module delivers the binary test message to the ToE.

In order to follow series of messages between the Input generator and the ToE, all the responses from the ToE will go through the same steps from Capturer through Parser, Protocol Logic, Test Logic and Serializer to Dispatcher. This way Flinder can test complete protocols and the Test Logic can take into account the responses of the ToE in order to generate the next test vectors.

² MSDL stands for Message Structure Description Language.

3.2 White-box testing

White-box testing mode is very similar to black-box testing. In this case the operation is still test-based, that is Flinder will execute the ToE several times differently injecting faults into its internal state. To do this Flinder will insert itself not into the communication between the Input generator (which is still needed) and the ToE, but into the function calling mechanism inside the ToE. The source code will be modified by inserting special hooks into the tested function, so Flinder can observe and modify input parameters of such hooked functions. The following tasks should be solved for this approach to work:

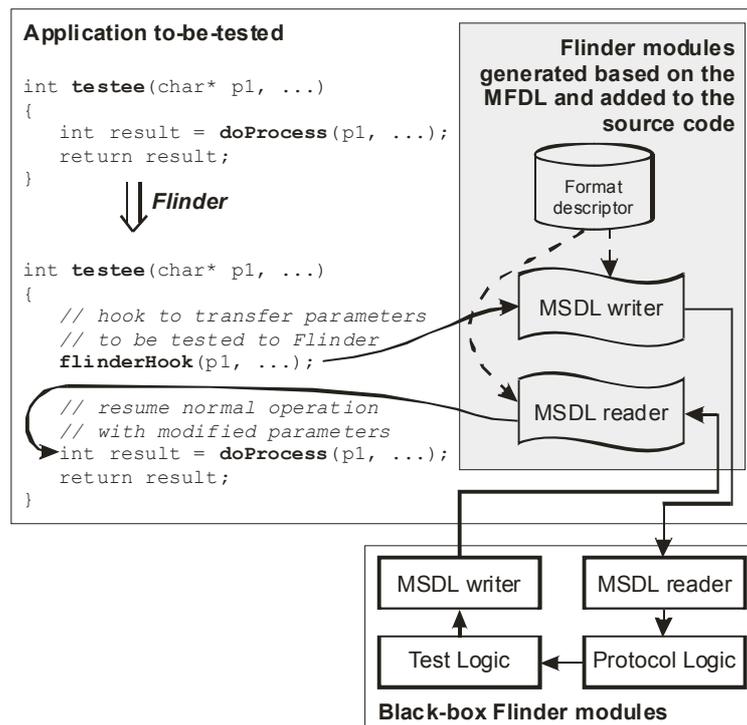


Figure 2 – White-box testing with Flinder

1. In white-box case the message format descriptors will be created to define the structure of input parameters of functions. It is possible to derive the MFDL structure semi-automatically from the source code (e.g. from C/C++ type definitions), however correlations between the fields of structures need to be expressed similarly to the black-box testing scenario in *actions*, which may need to be specified manually. (An example is depicted in Figure 3 – in the upper left corner a C-style variable declaration and some type definitions are illustrated, whereas in the upper right corner the corresponding MFDL is shown with the necessary actions).
2. The next step is to indicate to Flinder, which methods and functions need to be tested and which variables (e.g. function parameters, global variables etc.) need to be modified. These indications can be inserted into the source code by the test engineer.
3. By pre-processing the source code, first a *hook* has to be implanted into the tested functions to pass the input parameters to Flinder and let it modify internal states of the tested application (this is similar to the tasks of the Capturer and Dispatcher in the black-

box scenario). Furthermore, the MFDL needs to be transformed into a function, which is capable of creating an MSDL from the tested variables and vice-versa (this functionality can be compared to parsing and serializing in the black-box scenario). The obtained MSDL is then passed to the black-box part of Flinder, to the Protocol Logic and to the Test Logic. Creating the source code portion that transforms the variable or class instance into an MSDL is done automatically based on the MFDL: for each compound type a method is generated that recursively processes all its children, whereas for primitive types Flinder provides default implementations. The actions specified in the MFDL are automatically copied into the appropriate functions. This step is depicted in Figure 3 in the lower part.

4. After the black-box part of Flinder has received the to-be-tested MSDL, the Protocol Logic updates the statechart of the protocol according to the event occurred. Next the Test Logic lets the actual test algorithm modify the MSDL.
5. Similarly to parsing, the serialization (i.e. transforming the MSDL back to an object or variable instance of the ToE) is done by a source code fragment that was automatically generated based on the MFDL (see Figure 3).

This approach enables Flinder to inject faults into any internal states of the tested application and check whether in any circumstance security-relevant strange behaviour (e.g. buffer or integer overflow or crash bug can occur). We suppose that in a fail-safe application none of the functions should behave strangely regardless of inputs they get. Failing this assumption leads to security issues in most cases. So white-box Flinder tests can discover unhandled input states even inside an application, which are usually the sources of security flaws.

This approach was created in order to be able to re-use a great portion of Flinder created for black-box purposes. It can clearly be seen that the same Protocol Logic and Test Logic can be used for in both black- and white-box scenarios, these core modules operate on generic MFDL and MSDL data structures.

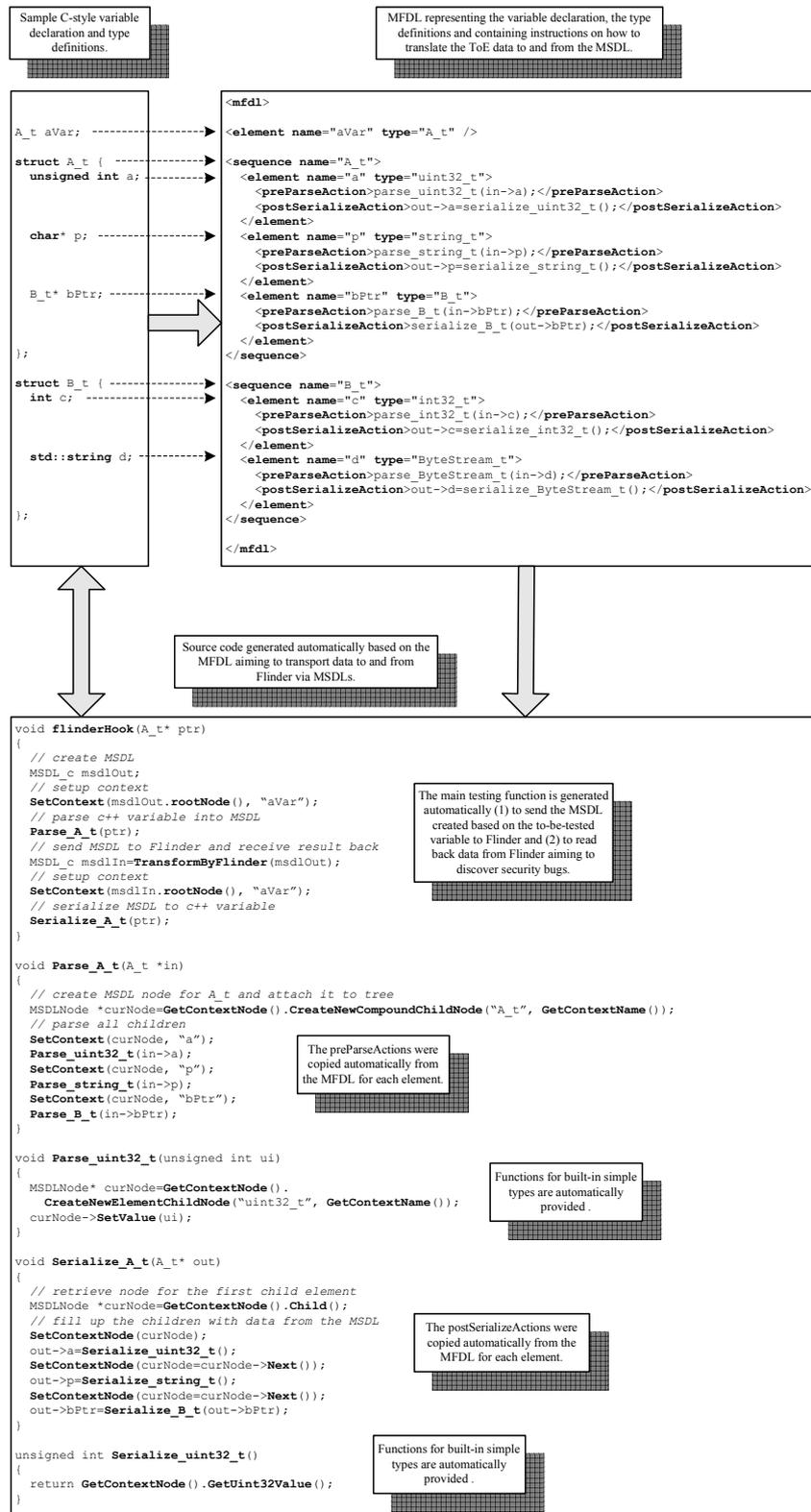


Figure 3 – Sample data structure, type definition with corresponding MSDDL and generated source code

4. FLINDER AT WORK

In order to illustrate the operation of Flinder, in this Section a practical example will be given. For this purpose we have “re-discovered” a buffer overflow type bug in widely used e-mail client application, the PINE version 4.56 [PINE-BOF].

```
MIME-Version: 1.0
Content-type: multipart/mixed; boundary="boundary"
Subject: hello

--boundary
Content-type: message/external-body; access-type=URL; A=1
Attrib: value
Message
```



Buffer overflow occurs if an attribute name in the 'message/external-body' header is longer than 20k.

Figure 4 – Message structure to be tested with Flinder

Figure 5 illustrates the input file format (MIME [RFC-MIME]) which is used by PINE. The actual overflow occurred when a 'message/external-body' header was present and an attribute name was longer than 20kB. In this case this oversized attribute name could overwrite control structures of the application and redirect execution even to the just injected buffer, thus allowing arbitrary code execution³.

4.1 The MFDL

An illustrative fragment of the MFDL created for the MIME message is illustrated in Figure 6. The main characteristics are:

³ We will not go into details about the buffer overflow bug itself as it is not the scope of this document. Detailed description and exploitation techniques are discussed in [BOF].

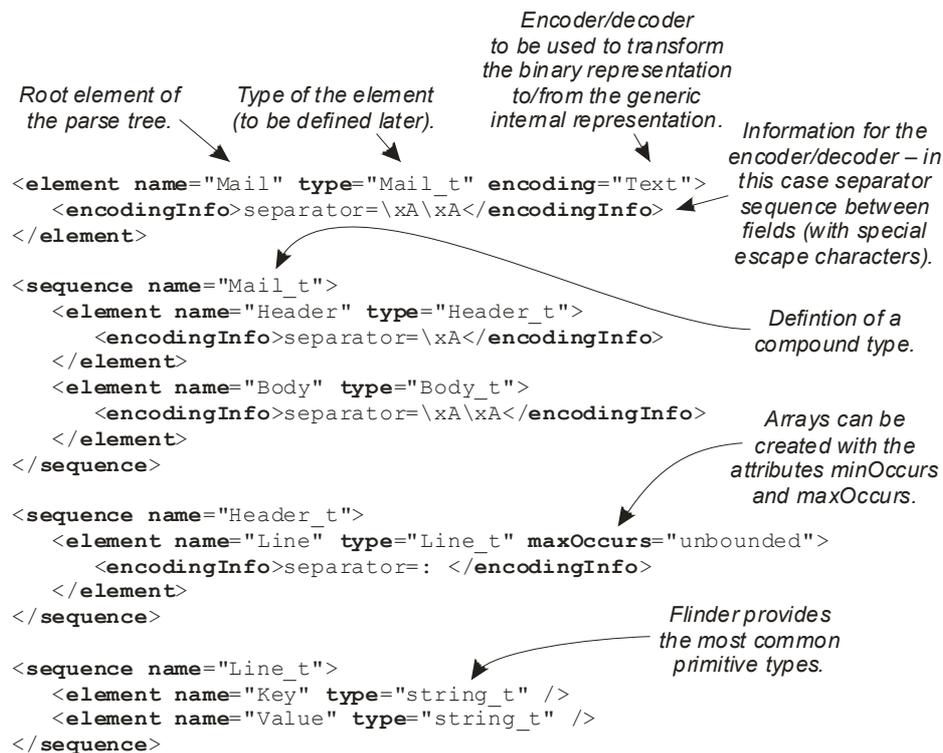


Figure 5 – Example MFDL

- The MFDL introduces the potential root nodes of the parse tree as elements. In this case we chose a text-based encoder, where the fields of the root node were separated by <LF><LF> (i.e. separator between header and body).
- After the definition of the root elements, the MFDL continues with the definition of the different compound types, which in this particular case are sequences. Besides sequences MFDL supports also choices (i.e. where only one element of the listed ones can follow, in contrast to the sequence where all must follow in the given order).
- The definition of the header illustrated the possibility of creating lists of similar items with the constructs ‘minOccurs’ and ‘maxOccurs’.
- Finally, Flinder defines the primitive types, which can be used in leaf nodes. It is the responsibility of the different decoders to transform the binary representation of such types into the generic format (MSDL) processed by Flinder.

Based on this MFDL, the generic buffer overflow testing algorithm could create test vectors that successfully triggered the buffer overflow in PINE.

4.2 Finding the buffer overflow

The buffer-overflow testing algorithm uses the generic buffer model (see Figure 6). In this model buffers have three properties: (1) they have an allocated length to which write access will be *successful*, (2) there might be length checking mechanism, which *rejects* out-of-range input buffers, and (3) the control might be inaccurate and there might be a range, which

results in *erroneous* operation. The goal of the test algorithm is to identify for each buffer-type field in the tested messages these three intervals and alert if the variable length error-interval exists.

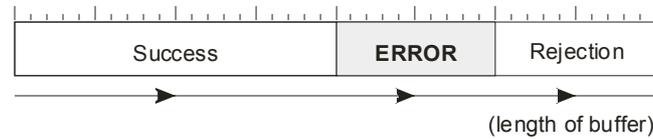


Figure 6 – Generic buffer model

In order to find out the properties of each buffer the test algorithm (see Figure 7) works as follows (for simplicity first let us assume only one buffer): (1) the algorithm enlarges the size of the buffer to twice the previous length as long as the ToE either rejects the input or an error happens; (2) if an error happened the algorithm already found the erroneous interval; (3) if a rejection happened, then via successive approximation smallest rejected and biggest accepted sizes are halved; (4) if these two values (i.e. the ranges of the accepted and rejected intervals) meet then the buffer is considered safe; otherwise (5) an error happened and similarly to (2) the erroneous interval is found.

Algorithm 1 Generic testing algorithm for one buffer

```

lastAccepted ← buffer.length
lastRejected ← -1
errorLength ← -1
while (errorLength = -1) ∧ (lastRejected ≠ lastAccepted + 1) do
  if lastRejected = -1 then
    buffer.length ← buffer.length · 2
  else
    buffer.length ← ⌈  $\frac{\textit{lastAccepted} + \textit{lastRejected}}{2}$  ⌉
  end if
  ⇒ Adjust the buffer according to the length and test the message with the ToE
  if ToE accepted the message then
    lastAccepted ← buffer.length
  else if ToE rejected the message then
    lastRejected ← buffer.length
  else if Error happened then
    errorLength ← buffer.length
  end if
end while

```

Figure 7 – Generic testing algorithm for one buffer

Naturally the algorithm can be generalized to test multiple buffers at the same time and thus speed up the testing process. However, in this case one has to consider the correlation between different buffers, i.e. a certain bug may not be revealed if a specific buffer does not contain a given value (e.g. if the content type of the body is not ‘message/external-body’, then the attributes will not be parsed accordingly and the buffer overflow will not be triggered).

4.3 Testing in source code

In this section we will show that the algorithm introduced in the previous section in connection with black-box testing can also successfully be used for white-box testing. For this purpose let us continue based on the type definitions shown in Figure 3 and assume the following erroneous example function, which contains also a buffer overflow:

```
void ErroneousFunction A_t *aPtr
{
    // Testing with Flinder
    flinderHook aPtr ;

    // 10-byte-long local buffer
    char locBuf[10];

    // If aPtr->p is longer than 10 bytes,
    // then this copying can overwrite the
    // return address of this function.
    // stored on the stack due to the missing
    // length checking, which results in
    // a security bug
    strcpy locBuf, aPtr->p ;
}
```

Figure 8 – Function containing a buffer-overflow-type vulnerability

If during the execution of the above function the `p` field of the `A_t` structure points to a string longer than 10 bytes then the copying will overwrite the fix-sized local buffer. Under certain circumstances this overwriting may reach control structures stored on the stack, and can even modify the return address of the function. This way an attacker may redirect program execution after returning from the function. Should we overwrite the buffer with non-malicious random values then with very high probability the application will cause a protection fault.

During testing the code portion generated by Flinder will first transform the `A_t` instance to MSDL, submit it to Flinder and after the modifications the MSDL will be serialized back to the variable. After this variable manipulation the normal operation of the ToE will be resumed and the reactions will be monitored. This way we can use the previously introduced algorithm in the Test Logic to search for buffer overflow type bugs, by iteratively enlarging string buffers in the MSDL (i.e. the `p` field of `A_t` and the `d` field of `B_t`) the algorithm will sooner or later create an MSDL which contains a long enough `p` field to trigger the protection fault detectable by Flinder. This way Flinder can pinpoint missing or inadequate length checking in the source code which could lead to buffer overflow type vulnerabilities.

With this simple example we could demonstrate the main advantage of Flinder: reusable test algorithms working on the generic MSDL and MFDL data structures. Flinder incorporates several such methods that can discover programming bugs of several generic types. By using these algorithms to find the security-relevant programming bugs, the security properties of the ToE can significantly be strengthened.

5. CONCLUSION

In this document we presented the Flinder approach for localizing security-relevant programming bugs in a semi-automated fashion. The novelty of this technique lies in the simplicity of the approach used – Flinder only requires correct input messages, message format descriptors and a statechart of the tested protocol in order to create and automatically dispatch test vectors to the ToE. For the creation of these test vectors Flinder uses generic test algorithms that work on the internalized generic structure of the messages that were created based on the MFDLs.

Flinder currently supports test algorithms for the most common security bugs, one direction of future work is to implement test algorithms for an even wider variety of bugs. Another important aspect is the integration into existing, deployed test frameworks. Finally, the capabilities of white-box testing should be extended in order to use the information present in the source code to deliver specific test vectors aiming to reveal security bugs.

6. REFERENCES

- [BOF] Aleph One: Smashing the stack for fun and profit, Phrack 7, 1996
- [PINE-BOF] iDEFENSE Labs: iDEFENSE security advisory 09.10.03: Two exploitable overflows in PINE, 2003
- [RFC-MIME] Fred N., Borenstein N.: RFC 2045 – Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies, 1996
- [SCHEMA] W3C World Wide Web Consortium: XML Schema, 2004
- [UML] Object Management Group (OMG): UML – Unified Modelling Language (UML), 2005