Practical Automated Filter Generation to Explicitly Enforce Implicit Input Assumptions

Valentin Razmov
University of Washington
valentin@cs.washington.edu

Daniel R. Simon
Microsoft Research
dansimon@microsoft.com

Abstract

Vulnerabilities in distributed applications are being uncovered and exploited faster than software engineers can patch the security holes. All too often these weaknesses result from implicit assumptions made by an application about its inputs. One approach to defending against their exploitation is to interpose a filter between the input source and the application that verifies that the application's assumptions about its inputs actually hold. However, ad hoc design of such filters is nearly as tedious and error-prone as patching the original application itself. We have automated the filter generation process based on a simple formal description of a broad class of assumptions about the inputs to an application. Focusing on the back-end server application case, we have prototyped an easy-to-use tool that generates server-side filtering scripts. These can then be quickly installed on a front-end web server (either in concert with the application or when a vulnerability is uncovered), thus shielding the server application from a variety of existing and exploited attacks, as solutions requiring changes to the application are developed and tested. Our measurements suggest that input filtering can be done efficiently and should not be a performance concern for moderately loaded web servers. The overall approach may be generalizable to other domains, such as firewall filter generation and API wrapper filter generation.

1. Introduction

The majority of recent security attacks have exploited various unwarranted assumptions about input data passed to server applications. Among these vulnerabilities, buffer overflows have accounted for over 50% of the security breaches [9, 25, 22, 24]. While simple to avoid, this type of attack has seen such proliferation mainly because of the typical programmer's "natural" disposition not to expect the unexpected, coupled with the relative ease – especially magnified through attack scripts [1] – with which malicious parties can exploit these vulnerabilities [10].

Other input anomalies often responsible for vulnerabilities include:

- parameters having unexpected values, especially values outside of (expected) ranges;
- ambiguous character string parameters that cause data to be misinterpreted as execution code [4]. Often these incorporate special escape characters that have different meanings in different contexts.

In all of the above cases, clearly and explicitly stating and verifying all implicit assumptions about the inputs is the key to trouble avoidance. However, it is unrealistic to expect that application developers will always include 100% of the necessary validation checks on the inputs their applications receive. For example, many existing applications were originally written under the assumption that they would run in trusted environments, and so lack needed input validity tests when used in environments vulnerable to attack. The option of retrofitting security into each legacy application's code in such circumstances is unrealistic. More generally, typical programming methodology neither requires nor helps the programmer to express input validity assumptions, making it easy to fail to verify them at all and requiring great discipline to get the verification right.

One approach is to externalize validation checks into a filter and then install the filter (only) in environments where it is necessary. This decouples input sanitization from core application functionality, thereby making it possible to develop an input filter long after its targeted application has been disseminated, without the necessity to have access to the application source code, and with much less effort as compared to developing a security patch. The inherent difference in complexity between simple input filters and full-fledged applications allows assumptions about the input to be specified quickly, compactly and in a less error-prone manner. (For example, a simple filter can be developed and deployed as soon as a vulnerability is discovered, and removed later when a patch becomes available and is installed.) In addition, filters external to applications allow for efficiency-minded omission of input validation if its presence is unnecessary.

The decision of when performing input validation is worthwhile can then be left up to the system administrator, who can decide if the performance overhead from the validity checks is outweighed by the need for protection in environments which cannot be guaranteed to be secure.

This approach is already the norm in many settings. For example, some applications perform no authentication or input validation, implicitly assuming that packets reaching them at a certain port always originate from senders inside their LAN. In this case, packets directed at that particular port but originating from outside the LAN will be treated, erroneously, as local – a situation which has not been anticipated by the application and is thus a possible cause of a security breach. Similarly, server applications sometimes implicitly assume that clients never send inputs large enough to overrun the available buffer space. In this scenario, the server application is prone to buffer overrun attacks, which have been shown [25, 9] to open the door to execution of arbitrary (untrusted) code on the server.

In both cases there are simple ways to prevent potential break-ins. For the first, a firewall filter can intercept and "filter out" packets coming from outside the LAN and directed at the port on which the application is listening. For the second, an input filter could check if the size (among other relevant characteristics) of the input is within an acceptable range and treat the input as unsafe otherwise. In both situations, employing such a filtering approach would ensure that incoming data conforms to the applications' inherent assumptions about it [10].

Of course, complete verification of the validity of an application's input can be as complex as the application itself. However, a substantial fraction of security vulnerabilities (such as the buffer overruns mentioned above) can be prevented by filtering inputs based on a few simple, syntactic properties (e.g., input length or the presence of certain "forbidden" characters or character sequences). These vulnerabilities are typically the first target of attackers, since they are relatively easy to exploit, requiring little understanding of the internal semantics of a given application. Moreover they are a common phenomenon, given the prevalence of various error-prone programming practices. The approach we take reflects our belief in security through simplicity as advocated in [18]; the more complex a filter becomes, the less trustworthy it tends to be.

Although not all possible vulnerabilities can be addressed by applying purely syntactic filters, the bar for breaking into a system merely by sending it invalid inputs can thus be raised significantly. Syntactic filters have long been used to stop malformed or otherwise unsafe incoming data from wreaking havoc on vulnerable server applications, and in firewalls to protect networks from external (or unauthorized internal) attackers subverting network-aware applications by masquerading as internal (or authorized) users. However, filtering has been very much an ad hoc practice so far, with administrators having to manually develop scripts, each specially tailored for a given application that needed shielding. While relying on the specifics of an application and protecting it as a separate entity helps build the most resilient of defenses, we believe that too much unnecessary burden has been placed on administrators, who have had to write intricate *ad hoc* scripts (i.e., filters) and support them over time.

Automating the filter generation would enhance the process immensely, reducing both the cost of creating and maintaining filters and the chance for occasional scripting errors or omissions. With the burden of managing filters thus reduced, the reliability and responsiveness of system defenses should improve, making them more resilient to attacks.

The next section provides an account of related work. Section 3 discusses the design decisions behind building filters and describes our software architecture. Section 4 concentrates on the implementation details of a proof-of-concept filter-generating tool, while Section 5 provides an evaluation of our experience with writing filters using the tool and a performance cost estimate of input filtering on real servers. We conclude in Section 6 stating the contributions of our work and outlining further research avenues.

2. Related Work

In [17] the authors deal with analyzing applet bytecode before applet instantiation in order to verify that applets conform to a set of filtering rules designed to preclude exploitation of known bugs in the JVM implementation. Rule-based filtering is proposed there (as well as in [12]) as an approach to patching bugs and preventing breaches. Our prototype filters have a similar goal, but in the back-end server context. We also enforce a set of assumptions on the application input, rather than on the application itself.

Clearly for any tool to be practical, usability needs to be one of its primary goals. This requirement was a major factor in our design. An approach similar in spirit is taken in [3], where a generic representation scheme for firewall filter configuration is presented, leading to a usable tool for firewall management. In contrast, in [23] the emphasis is on generality and formal assurance, with the result that filters are expressed in a rich (and heavyweight) filter specification language, as opposed to our simple one. We hope tools like our prototype will make generating back-end server input filters simpler, more intuitive and less error-prone.

Another goal we have set for our tool is to avoid relying on access to source code, which is often unavailable in practice. This distinguishes our approach

from most other previous work, including [17], [12], [11] and [25]. The decision is a trade-off between wider applicability and the ability to do (offline) syntactic as well as limited semantic analysis of the application's inner workings, such as looking for spots where (not necessarily maliciously intended) input could "trip" on a bug, resulting in a potential security hole. The authors of [14], like us, do not rely on source code inspection in their attempt to contain untrusted helper applications. They too set it as their goal to keep the security mechanism orthogonal to any non-security-critical functionality. However, they use "modules" to enforce a resource sandboxing policy, rather than an input filtering policy. Hence, their modules are based on a sophisticated understanding of the resources available on a user's machine, their security criticality, and their legitimate uses, and are appropriate for protecting against viruses or Trojan horses being downloaded and run as applications. Several other "wrapper" designs have been proposed for the same general sandboxing task [2, 16, 20], or for network security policy enforcement [19], but they have all had generality of policy expressiveness as their main goal, and are thus relatively complex and difficult to apply. Our filters, on the other hand, are meant to protect against simple (yet still potentially highly damaging) malformed inputs, and thus should be much easier to write and apply.

3. Design

Next we discuss the design decisions behind our automated filter generator.

3.1. Architecture

To be effective, a filter needs to be specific to an application. For any given application, the process of generating a filter requires that at some point the assumptions about the application's input be expressed in a formal way. If the application's internals (i.e., source code, etc.) are undisclosed, then the formal description can be provided by the application's vendor or by a trusted third party. Alternatively, an administrator can generate it (using, for instance, a GUI-based tool like the one from our prototype) based on the application's specification, and/or on descriptions (formal or informal) of particular attacks the filter is meant to block.

The full process of filter generation is illustrated in Figure 1. The components are:

- Intermediate Description Synthesizer – a combined (graphical user interface-based) description synthesis tool, used to generate, in order, a formal description of the input structure (defining how the input is to be parsed into a sequence of tokens – we refer to them below as "parameters") and a formal description of the assumptions being enforced on these parameters. Either or both of these descriptions could also be generated manually, or by means of other tools, or obtained from trusted vendors. However, the description synthesizer may improve usability and may reduce potential user errors.
- Filter Generator – a filter-generating tool, which "compiles" the above two formal descriptions into an executable filtering script.

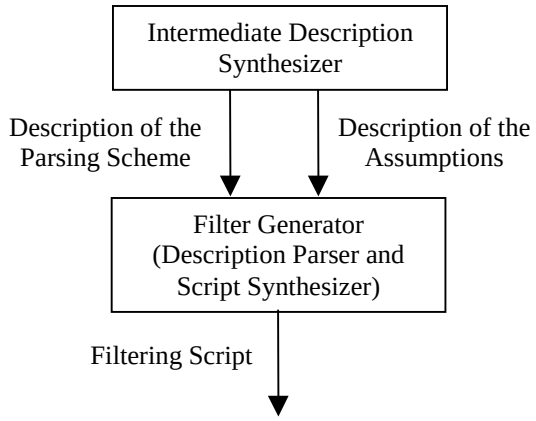

Figure 1. Filter generation process

At run time, the application's input is intercepted (by a special module) and routed to the filtering script. The script, which encodes the class of valid input structures, parses the given input into parameters and then tests the compiled assumptions on them. Once the checks are completed, the intercepting module receives the script's verdict and takes the appropriate action – passing the input on to the application or dropping it (Fig.2).

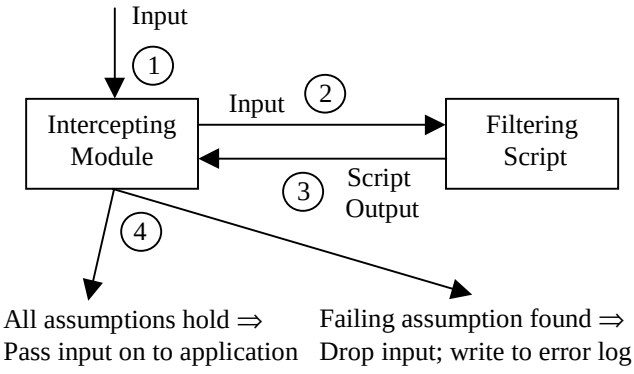

Figure 2. Decision-making process – events are numbered in temporal sequence

Note that the script generation process (Fig.1) is completely decoupled from the real-time processing of input data (Fig.2), i.e., the filtering script is generated entirely offline, once per application. The performance cost of filtering is thus determined solely by the cost of script interpretation at run-time, plus the time to perform

two local procedure calls – from the intercepting module to the filtering script and back.

3.2. Expressing Assumptions

We express each assumption about an application's input as a logical clause. Semantically these clauses are AND-ed together to form a Boolean expression in conjunctive normal form (CNF). Verifying that the Boolean expression is satisfied for a given input is equivalent to ensuring that all stated assumptions on this input hold. For a broad set of assumptions, their representation naturally translates into simple logical expressions without disjunction or negation operations; this in turn significantly simplifies the formal language for describing assumptions. There are few exceptions requiring more complex clauses and the semantics of a disjunction (we discuss them in the Implementation section); as it turns out, these can be easily handled without compromising the simplicity of the formal description.

An important design choice we made was to embrace the "positive" filtering approach (as in [21]), having assumptions explicitly specify what is permitted, rather than what is not. (We allowed one specific exception to this rule, adding a provision for checking for known rogue inputs. Although such an assumption carries a "negative" meaning, it does not alter the general thrust toward positive filtering, while significantly enhancing the expressive power and usefulness of filters, as explained in [4].) In contrast, [17] specifies known misbehaviors on the part of a Java applet, and applies a set of filtering rules to each new (downloaded) applet. In [12] too, incorrect – rather than correct – behavior is specified in the security policy, which restricts the violations to check for to only a set of well-known and anticipated attacks. The default in both of these systems is to allow any behavior not explicitly prohibited. Our more conservative approach has the benefit of making filters more likely to catch unknown vulnerabilities as well (for which a "negative" filter can only be specified after the fact). Moreover, in the case of an underspecified input filtering policy, the error would more often be on the side of security.

3.3. Intermediate Description Language

For greater flexibility we decided in favor of going through intermediate filter descriptions and against generating filtering scripts directly from a GUI-based tool (which would have effectively merged the description synthesizer and filter generator in Figure 1). As a result, filter specification is decoupled from script generation, allowing the administrator to mix choices of specific tools to be used for each.

The most obvious candidate for an intermediate description language is XML [26] – an emerging standard for data representation in distributed applications. XML ideally fits our needs, being flexible (allowing custom-defined structures), extensible, human-readable and portable across platforms. The portability in particular ensures that the subset of XML we use can serve as a common intermediate description language, while anything built on top can be replaced or bypassed.

4. Implementation

For our prototype implementation, we focus on the back-end server application case. In a typical scenario, a web server receives HTTP requests from clients and passes those on to back-end server applications. If requests (say, in the form of scripting page URLs followed by input values presented as tuples of the form '`<attribute>=<value>`') are not validated, then maliciously constructed data in the requests (e.g., values of unexpectedly large sizes, or ones containing active scripting code disguised as data, etc.)may undermine the operation of the back-end server application. Hence the need for filtering of client input data by web servers.

4.1. Web Server Input Filtering

Once a web server filtering script is produced by a filter generator (via the process illustrated in Figure 1), input filtering works as follows (see Figure 2):

- Upon receiving an HTTP request, the filtering script is invoked by the web server via an intercepting module. (For example, in the case of an IIS server running on Windows NT, the filtering script is registered as a COM object, and the intercepting module is an ISAPI filter DLL configured to call the script via its COM interface.)
- The intercepting module passes the request to the filtering script, which then parses it into a set of "parameters", according to the corresponding XML description of the parsing scheme (as implemented in the script).
- The filtering script tests each extracted parameter to make sure it satisfies the assumptions specified in the corresponding XML description of assumptions (and thus also implemented in the script).
- The output of the filtering script is an error code indicating whether the HTTP request was accepted by the filter and, if not, what problem was detected. The intercepting module receives that output and either passes the HTTP request on to the back-end application (if the request was successfully validated) or writes to an error log and has the web server redirect the client to an error notification URL (in the case of a rejected HTTP request).

Next, we give implementation details for the building blocks shown in Figure 1.

4.2. Scripting Language

The appropriate scripting language necessarily depends on the type of filter being generated. For instance, if a firewall filter were needed, a specially tailored (firewall) scripting language would be employed, whereas in the context of back-end server applications it would be a scripting language suitable for web servers, such as VBScript or JavaScript. We have chosen VBScript [15] for our prototype implementation. However, we note that, in order to provide a proof-of-concept solution, our filtering scripts do not use any language-specific or platform-specific features. Thus, the scripting language itself may later be replaced by altering the component of the filter generator responsible for writing out the filtering script, without necessitating any changes to other components.

4.3. Description of the Parsing Scheme

The input is initially treated as a single character string, which is to be delimited first into what would become the constituent parameters. As an example, consider the sample HTTP request shown in Figure 3 being sent to a web server, and one possible parsing of this input string into parameters.

We allow two alternative ways to define the boundary between one parameter and the next: by specifying a single-character end-delimiter of the current (first) parameter, or by giving the exact length of that parameter. The only exception to this rule is the very last parameter – running to the end of the input string – it is "delimited" by the special `NULL` end-delimiter. (An end-delimiter character, if specified, is excluded from the current extracted parameter.)

If there are multiple possible formats for the input (with different structures and thus different constituent parameters), then a separate filter will be necessary for each such structure. The intercepting module would invoke each filter, and accept the input only if it is accepted by at least one of the filters.

To aid the administrator in formally describing a parsing scheme, we have implemented a GUI-based tool, the output of which is the XML description of the parsing scheme specified via the GUI. A sample XML description is shown in Figure 4.

It highlights the small uniform set of XML tags needed to express how a string is to be parsed into parameters. Each parameter has its boundaries defined inside an instance of a `<param>` tag structure. This structure represents a collection of four substructures:

- `<number>` – the current parameter number (in order from left to right);
- `<function>` – specifies whether the end of the current parameter is determined by an exact parameter length in number of characters (`LENGTH`) or by an end-delimiter character (`END_DELIM`);
- `<value>` – either the parameter length or the end-delimiter character, depending on the choice of `<function>`;
- `<desc>` – (optional) concise natural-language description of the semantics of the current parameter.

```
<input-parse>
  <param>
    <number>1</number>
    <function>END_DELIM</function>
    <value>:</value>
    <desc>Protocol name</desc>
  </param>
  <param>
    <number>2</number>
    <function>LENGTH</function>
    <value>2</value>
    <desc>Double slash, i.e. //</desc>
  </param>
  <param>
    <number>3</number>
    <function>END_DELIM</function>
    <value>/</value>
    <desc>Server name</desc>
  </param>
  <!—- The definitions of parameters 4-7 have
       been omitted for brevity -->
  <param>
    <number>8</number>
    <function>END_DELIM</function>
    <value>NULL</value>
    <desc>Second (real) parameter value</desc>
  </param>
</input-parse>
```

Figure 4. XML parsing scheme description of the sample HTTP request; output from GUI tool

Reading the XML description in Figure 4, parameter 1 (semantically, the protocol name) starts from the beginning of the input string and ends immediately before the first colon. Then, parameter 2 (meant to be the delimiting double slash after the protocol name) picks up immediately after the delimiting colon and is exactly 2 characters long. Parameter 3 (supposed to contain a server name) starts immediately after parameter 2 and continues until the next slash-character in the input, etc. Parameter 8

```
        http://www.foomusic.com/my-query?page=%2Fpop%2F&club=classical
http[1] //[2] www.foomusic.com[3] my-query[4] page[5] %2Fpop%2F[6] club[7] classical[8]
```

Figure 3. One possible parsing of a sample HTTP request string into parameters. For clarity parameters (second line) are shown separated by spaces and numbered in superscript

(meant in our sample break-up to hold the value of the "club" property) starts (naturally) after parameter 7 and is comprised of all characters until the end of the input string.

4.4. Description of the Assumptions

We now turn to the set of assumptions that can be enforced on each parameter.

The majority of desirable constraints can be expressed using a fairly simple set of relations. The expected improved usability was the reason we decided against expressing assumptions using more general and powerful languages such as that of regular expressions. Although there may be a few constraints of practical interest that regular expressions, but not our limited language, can capture, we have not sacrificed any essential functionality (although some assumptions may need to be expressed as "complex assumptions"; see below). Our goal is to be able to prevent a substantial fraction of the recurring (and potentially highly damaging) vulnerabilities, while keeping the filtering process as simple as possible. Avoiding some security holes may require more complex treatment, but many others will be relatively easy to fend off. (Facing a similar choice between simplicity and extra functionality the authors of [21] also decided in favor of simplicity.) We believe that in practice this more than makes up for the lack of extra generality in the basic language; the simpler the security tools, the more likely it is that they will be used effectively.

Our tool implements a set of 13 relations (Table 1). Adding more would not conflict with the model; however it would require minor changes to the filter-generating tool.

In the XML description of assumptions, each assumption is specified either on a parameter itself (SELF) or on its size (SIZE). (The latter naturally takes only numerical values.)

Consider the sample HTTP request (above) and the set of parameters extracted from it. Below is a partial list of the assumptions (first in natural language, then in XML) that one might wish to impose on some of the parameters:

Parameter 1 (protocol name) should be one of "http" or "https"

Parameter 3 (server name) should be no more than 30 characters long

Parameter 3 (server name) should have ".com" as suffix

We describe these sample assumptions in XML in Figure 5. As the example makes obvious, each assumption on a parameter is described within a separate <param> tag structure. That structure is comprised of six substructures:

- <number> – the number (in order from left to right) of the parameter to which the assumption applies;
- <func> – the parameter function, specifying whether the current assumption concerns the parameter itself (SELF), or its size in number of characters (SIZE);
- <cond> – the relation (from Table 1) between the current parameter (or its size – depending on the choice of <func>) and the list of constants specified in the <value> tag;
- <delim> – the single character that delimits constants in the <value> list (if the list contains a single constant, a NULL delimiter is specified);
- <value> – the list of one or more constants being related to the current parameter or its size (depending on the choice of <func>);
- <desc> – (optional) concise natural-language description of the semantics of the current assumption.

```
<param>
  <number>1</number>
  <func>SELF</func>
  <cond>LEX_EQ</cond>
  <delim>,</delim>
  <value>http,https</value>
  <desc>Parameter 1 should be one of the strings
        "http" or "https"</desc>
</param>
<param>
  <number>3</number>
  <func>SIZE</func>
  <cond>LE</cond>
  <delim>NULL</delim>
  <value>30</value>
  <desc>Parameter 3 should be no more than 30
        characters long</desc>
</param>
<param>
  <number>3</number>
  <func>SELF</func>
  <cond>ENDS_IN</cond>
  <delim>NULL</delim>
  <value>.com</value>
  <desc>Parameter 3 should end with the string
        ".com"</desc>
</param>
```

Figure 5. Sample XML description of assumptions on parameters

If the <value> list contains more than one constant, all subsequent constants have to be of the same type as the first one. For instance, the relation EQ expects one or more numerical values, all integers or all floating-point numbers. This underscores one of the advantages of using a GUI-based intermediate description synthesizer – automated type-checking at description generation time.

When there are two or more constants in the <value> list, the relation has disjunctive semantics: the parameter in question satisfies the specified relation with the list if it does so with at least one of the constants in the list. (See Table 1 and its footnote for a list of relations allowing disjunctive semantics.) For instance, the first

assumption in Figure 5 demands that parameter 1 be lexicographically equal to one of `http` or `https`.

It is clear from Table 1 that checking if a parameter is among a given list of values, and validating the allowable character set for the representation of a parameter, are the two situations when disjunction is involved in specifying assumptions. Both are easily handled in the implementation.

Finally, an input filter consists of a set of assumptions on the extracted parameters. The XML structure `<server-filter>` describes a back-end server filter.

```
<server-filter>
  <param-count>8</param-count>
  <max-total-length>100</max-total-length>
  <param-desc>See the XML parsing schema
              descriptions</param-desc>
  <!—- Definitions of assumptions on parameters
       are omitted here -->
  <complex-cond>
    <xml-file>MoreComplex.xml</xml-file>
    <cond-desc>Some more complex
               assumptions</cond-desc>
  </complex-cond>
</server-filter>
```

Figure 6. Sample XML description of a back-end server input filter

It specifies:

- `<param-count>` – the number of parameters;
- `<max-total-length>` – the maximum allowable length of the input string before it is parsed into parameters;
- `<param-desc>` – (optional) concise natural-language description of the filter semantics;
- one or more `<param>` structures (as discussed above), each describing an assumption on some parameter;
- zero or more `<complex-cond>` structures, each containing a "pointer" to an XML file and an optional description of it (Fig. 6).

Complex assumptions are a provision for making the model extensible by adding a level of indirection, which helps to avoid changes to the core XML format or the filter-generating code. They are specified in dedicated XML files, referred to by the `<xml-file>` tag inside the `<complex-cond>` structure (see Figure 6). Complex assumptions also allow additions to a filter format to happen incrementally (i.e., as more reports and exploits become known) and with ensured backward compatibility. Thus, assumptions which otherwise would not fit into our current structure can still be described, albeit in separate XML files. We do not specify what the structure of such files should be.

One potential practical use of complex assumptions is as follows: assumptions described without resorting to the `<complex-cond>` provision only involve relations between parameters and specific constant values (from the `<value>` substructure) – numerical values, character strings or character set ranges. While this is most often the case now, it may turn out to be useful to provide for future enhancements. An example of a complex assumption may involve computing a checksum over several network

Table 1. Relations used in defining assumptions on parameters

Relation name	Relational semantics
CONSISTS	The parameter as a string consists only of characters from a specified set, e.g., a-z, 0-9, etc. [1]
EXCLUDES	The parameter as a string excludes a specified substring [2]
ENDS_IN	The parameter as a string ends in a specified string (suffix) [1]
LE	The parameter or its size (whichever is specified) is less than or equal to a given numerical constant
LT	The parameter or its size (whichever is specified) is less than a given numerical constant
GE	The parameter or its size (whichever is specified) is greater than or equal to a given numerical constant
GT	The parameter or its size (whichever is specified) is greater than a given numerical constant
EQ	The parameter or its size (whichever is specified) is equal to a given numerical constant [1]
LEX_LE	The parameter as a string lexicographically precedes or is equal to a specified string
LEX_LT	The parameter as a string lexicographically precedes a specified string
LEX_GE	The parameter as a string lexicographically succeeds or is equal to a specified string
LEX_GT	The parameter as a string lexicographically succeeds a specified string
LEX_EQ	The parameter as a string is lexicographically equal to a specified string [1]

[1] These relations can have disjunctive semantics, i.e., there can be more than one specified constant (all delimited appropriately) on the right-hand side, and if so, the parameter is required to satisfy the relation with at least one of these constants.

[2] The only relation with a "negative" meaning. It is provided specifically for exclusion of particular character strings known to have been used in malicious ways (e.g., from released bug reports; see [4]).

packet fields and comparing the result to some other packet field which itself is not a constant.

The complex condition mechanism works as follows: given a routine (say, in VBScript) that can verify the desired complex assumption, the routine's name and its arguments are recorded in an appropriately designed XML structure. Then, that structure is embedded in a file listed as one that contains complex assumptions.

To facilitate the process of formally describing assumptions, we have implemented a second GUI-based tool, the output of which is the XML description of assumptions specified via the GUI. The tool performs type-checking on all fields at the time of entering new data. Type- and format-checking is also done before a description is loaded from an XML file.

4.5. Prototype Statistics

Our implementation consists of approximately 4000 lines of C++ code; of those almost 2400 lines are core functionality unrelated to GUIs. For the sample HTTP request the intermediate XML description files are 50 lines (describing the parsing scheme for 8 parameters) and 180 lines (describing a total of 21 assumptions), respectively. The resulting script is approximately 350 lines of VBScript code with its size linearly proportional to both the number of parameters and the number of assumptions to be enforced on them. (An input resulting in, say, 16 parameters and 80 assumptions on them would need a filtering script of about 700 lines of code.)

Since our design is conceptually simple and the implementation is small enough in size to actually be comprehensible, we believe that our process of filter generation (and hence the resulting filtering scripts) should be less prone to errors than more complex filter generation mechanisms or software patch development. Our filters should also be easy to test quickly and effectively prior to distribution and application.

5. Evaluation

We evaluate our prototype implementation by assessing the extent to which it improves security and by estimating its effect on performance.

5.1. Security

Security is inherently hard to measure. There is no universally accepted procedure by which to evaluate it. Still there are practical indications which can give insight into the potential resilience of a system to threats.

To assess how successful our tool is with preventing break-ins, we looked at widely known archives of recent vulnerabilities, publicly reported over the past 2 years [9, 22]. Among them we identified three categories: those against which simple lexical filters (such as the back-end server filters generated by our prototype) can help defend, those which can be successfully fended off using more general input filtering, and finally those which cannot be exploited merely via malicious inputs and for which input filtering itself will be insufficient as a remedy. Our results are summarized in Table 2.

We defined a vulnerability as "reparable using sophisticated input filtering" if a sufficiently complex input filter could protect against its exploitation, but any purely lexical input filter would result in either significantly many false negatives (missed invalid inputs) or significantly many false positives (rejected valid inputs). A recent example is a vulnerability [5] in which the MIME types of HTML data were deliberately set incorrectly in order to trick a browser into automatically opening MIME objects containing malicious content, and disguised as benign MIME types. Although an input filter to protect against such a vulnerability is feasible, it would need to perform sophisticated analysis to determine the correct MIME type and treat input data accordingly. A simple lexical filter is bound to fail at that task. (Other examples can be found in [7].)

The "non-reparable using input filtering" category combines cases where no amount of mere input filtering, however sophisticated, would lead to protecting the vulnerable application. Those are typically denial of service attacks [6] or vulnerabilities resulting from incorrectly implemented application functionality [8] (as opposed to just lack of input validation). One example of the latter concerns TCP initial sequence numbers, which are supposed to be generated at random (or at least pseudorandomly) in order to be unguessable, and thus to prevent spoofing of TCP connections. Some implementations, however, have recently been shown not to have the necessary cryptographic properties. Input

Table 2. Classification of vulnerabilities reported in the period 2000–2001 and applicability of input filtering to defending against them

Year	Total number of examined vulnerabilities	Reparable using simple lexical filters	Reparable using sophisticated input filtering	Non-reparable using input filtering
2001	35	23 (13)	7 (7)	5 (5)
2000	29	16 (8)	3 (3)	10 (10)

Numbers in parentheses show the number of independent (i.e., unrelated to each other) classes of reported attacks in each category.

filtering clearly cannot remedy this problem.

The examined public archives indicate that simple lexical filters can help to defend against a significant fraction of the reported attacks, many of which are unrelated to each other. Our tool covers 100% of those vulnerabilities reparable by simple lexical filters.

5.2. Performance

To evaluate performance, we compare systems running an application without an interposed input filter versus systems in which a filter is set up to intercept and analyze inputs before forwarding those which are safe on to the back-end application. Since the running time of a filter of the type we are considering depends on the size and complexity of its inputs as well as on the script implementation complexity of the relations in Table 1, and since the running time of an application may have no obvious relationship to the size and/or complexity of its inputs, our comparisons will be based neither on relative execution times between both scenarios, nor on execution times of the filter versus those of the application. Instead, our approach will be to provide an account of the steps through which control flow passes in each of the two scenarios, and estimate on every step the absolute overhead due to the use of one of our input filters.

The differences resulting from the use of input filtering are highlighted in Table 3. The italicized operations are the ones responsible for the overhead in the filtering scenario: redirecting inputs to and from the filtering script (corresponding to steps 2 and 3 in Figure 2) plus actual script interpretation time.

Since both the filtering script and the intercepting module reside physically on the same machine acting as the web server, the communication time between the two is comparable to the time required to perform a local procedure call. On modern commodity machines this amounts to a few dozen instructions, or an execution time in the range of 50-100 ns.

We experimented with a filtering script generated from 21 assumptions on 8 parameters, and designed to validate our sample HTTP request. Our measurements, performed on an 800MHz Pentium III with 256MB RAM running under Windows 2000 (with all other applications closed), showed script interpretation times ranging from 1.10 ms to 1.14 ms, with both the mean and median at 1.11 ms.

Based on these numbers, the total overhead of running our filter on inputs with roughly the complexity of the sample HTTP request is on average 1.11 ms per validated input. Included in that calculation are the average script interpretation time and two local procedure calls (from the intercepting module to the filtering script and back). Since scripts are small in size and do not involve disk accesses, script interpretation is CPU-bound. It is also the major overhead contributor when input filtering is performed, resulting in an upper limit of 9.0 requests per second per percent of allocated CPU power. Hence, if on average 200 requests per second hit a dedicated web server machine with the above characteristics, these requests can be validated by allocating 22% of the CPU cycles to running filtering scripts. Conversely, if 50% of the CPU power can be dedicated to filtering, that translates to servicing up to 450 requests per second. Increasing the sophistication of the filter (say, by doubling the number of parameters and assumptions) but also switching to a high-end server machine is unlikely to change that estimate by much. (We should note that the composition of relations in a filter does matter: `EQ`, the "cheapest" relation, is interpreted in 0.8μs on average, while `CONSISTS`, the most complex one, takes on average 106μs for reasonably sized inputs – a factor of over 100 difference in performance. Also, a test filtering script with a single parameter showed execution times between 9.6 and 21.4 times faster – depending on the composition of relations – than those of the filtering script with 8 parameters generated for our sample HTTP request. This supports our hypothesis of a roughly linear correspondence between the number of parameters in a filter and its execution time.) Note that had our filtering scripts been compiled rather than interpreted, input

Table 3. Control flow comparison between architectures with and without input filtering

	Web server	Filtering script	Back-end application
With input filtering	Receives input		
	Forwards input to filtering script	*Receives input*	
		Interprets filtering script on input	
	Receives result from filtering script	*Sends result back to web server*	
	Forwards valid input to back-end application		Receives input, *if valid*
Without input filtering	Receives input	N/A	
	Forwards input to back-end application		Receives input in all cases

Two operations on the same line signify the passing of a "message" from one component to another (e.g., the web server forwards the input – the "message" – to the filtering script and the filtering script receives it).

validation would have been much faster, and performance overhead thus greatly reduced.

To make a real-world comparison, high-volume commercial web servers were reported [13] in the last months of 2000 to receive an average of over 50000 requests per second. Such server farms make use of dozens of dedicated front-end web server machines, load balancing requests to the back-end machines. Hence, the rate of incoming requests each single front-end web server sees is not very different from the sustainable rate of servicing requests in our architecture where input filtering is continually performed.

Since front-end web server machines are typically CPU-bound (whether they are serving static or dynamic content), diverting a portion of the server's CPU power (say, to perform input filtering) would necessarily have an impact on the overall performance of the server. However, given the above performance data, the impact would be insignificant for web servers experiencing moderate loads.

6. Conclusions and Future Work

The main contribution of this work is the automation of the filter generation process. Administrators still have to write formal specifications of what the filters should do, but the error-prone and time-consuming step of implementing the actual filtering scripts is now automated given such formal descriptions. The latter can be written in a simple language and, if appropriate and desired, graphical tools can be employed to make the process even more transparent and intuitive.

One important application of our work is in providing express response to security breaches through the dissemination of filtering scripts while traditional security patches are still under development. This can save time, money and reputation for the vendor of the application under attack. In many cases, where a vulnerability is due to lack of simple input validation – and these have accounted for a large fraction of recent breaches [9, 22, 24] – it should literally take a security expert minutes (as our informal experience suggests) to identify the right set of parameters and assumptions on them in order to generate the necessary filtering script. Furthermore, this can be accomplished without knowing the details of the application's source code (unlike [11]) and even without having access to it (which is most often the case). Consequently, filters can be quickly created (using, for instance, a non-source-based filter-generating tool) and disseminated by trusted third-party security monitoring vendors, rather than solely by the application's vendor, as is the case with patches. Thus, future attempts to exploit the same vulnerability will be fended off reliably. In comparison, it may take weeks and even months (of analyzing, developing and testing) before a reliable patch to a widely deployed commercial software product is ready for distribution. In the case of legacy software, no longer supported by its original vendor, patches will likely never come out; however, filters would be easy and inexpensive to generate and distribute.

Our measurements indicate that performance overhead should not be a deterrent against using filters to validate application inputs, except perhaps in highly performance-critical settings.

To further assess how intuitive our prototype is, we plan to conduct user studies on a set of known vulnerabilities taken from public archives [24, 22, 9]. After explaining our approach and providing the necessary set of relations (Table 1), users will be asked to identify for each vulnerability the set of parameters and assumptions they would choose to have a filter enforce, if they were to assume the task of administrators.

It is possible that a solution similar to this one could be adopted for automatically generating firewall filters. Generalized firewall configuration languages have been proposed [3]; a logical further step might be to explore automating the generation of configuration scripts in such a language using GUI-based tools to formally describe simple firewall policies. The idea might also be put to use encapsulating existing API function libraries, in a simplified version of the basic approach of [14]. A specially designed wrapper filter would intercept calls to its library routines, do the necessary validation checks and only "forward" the calls to the corresponding routines if it is safe to do so. Whenever a vulnerability in a library routine is discovered, an appropriate wrapper filter could be generated and applied as a stopgap until an OS or application patch is later released.

In fact, application programmers could perhaps specify at development time the assumptions made by their code about its inputs, and generate the appropriate filters themselves. (In some cases it might even be possible to generate the descriptions directly from source code.) Later, an administrator could decide, based on the hostility of the environment, the sensitivity of the application, and the performance constraints on the system, whether a filter should be installed to improve security or omitted to avoid affecting performance.

Acknowledgements

The authors would like to thank Jon Pincus and John Zahorjan for their thoughtful comments at various stages of this work. Paul England assisted us in dealing with a few implementation details. Mariusz Jakubowski helped configure our filters to work on a real server. A special thanks goes to Stani Vlasseva for proof-reading multiple versions of this paper and suggesting numerous improvements. The anonymous reviewers helped improve the final draft of this paper.

References

[1] H. Browne, W. Arbaugh, J. McHugh, W. Fithen, "A Trend Analysis of Exploitations", In Proc. 22nd IEEE Symposium on Security and Privacy, Oakland, CA, May 2001, pp.214-229.

[2] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS Software with Generic Software Wrappers", In Proc. 20th IEEE Symposium on Security and Privacy, Oakland, CA, May 1999, pp.2-16.

[3] Y. Bartal, A. Mayer, K. Nissim, A. Wool, "Firmato: A Novel Firewall Management Toolkit", In Proc. 20th IEEE Symposium on Security and Privacy, Oakland, CA, May 1999, pp.17-31.

[4] "Malicious HTML Tags Embedded in Client Web Requests" and "Understanding Malicious Content Mitigation for Web Developers", CERT Coordination Center, Feb. 2000, http://www.cert.org/advisories/CA-2000-02.html, http://www.cert.org/tech_tips/malicious_code_mitigation.html, http://www.cert.org/tech_tips/malicious_code_FAQ.html

[5] "Automatic Execution of Embedded MIME Types", CERT Coordination Center, Apr. 2001, http://www.cert.org/advisories/CA-2001-06.html

[6] "Denial-of-Service Vulnerabilities in TCP Stacks" and "Multiple Denial-of-Service Problems in ISC BIND", CERT Coordination Center, Nov. 2000, http://www.cert.org/advisories/CA-2000-21.html, http://www.cert.org/advisories/CA-2000-20.html

[7] "Multiple Vulnerabilities in Alcatel ADSL Modems", "Interbase Server Contains Compiled-in Back Door Account", "Input Validation Problems in LPRng" and "Microsoft 'IE Script'/Access/OBJECT Tag Vulnerability", CERT Coordination Center, 2000-2001, http://www.cert.org/advisories/CA-2001-08.html, http://www.cert.org/advisories/CA-2001-01.html, http://www.cert.org/advisories/CA-2000-22.html, http://www.cert.org/advisories/CA-2000-16.html

[8] "Superfluous Decoding Vulnerability in IIS", "Statistical Weakness in TCP/IP Initial Sequence Numbers" and "PGP May Encrypt Data with Unauthorized ADKs", CERT Coordination Center, 2000-2001, http://www.cert.org/advisories/CA-2001-12.html, http://www.cert.org/advisories/CA-2001-09.html, http://www.cert.org/advisories/CA-2000-18.html

[9] "CERT/CC Advisories 1988-2001", http://www.cert.org/advisories/ ; Relevant exploits in: "Two Input Validation Problems in FTPD" and "Input Validation Problem in rpc.statd", CERT Coordination Center, Jul-Aug. 2000, http://www.cert.org/advisories/CA-2000-13.html, http://www.cert.org/advisories/CA-2000-17.html

[10] P. Chowdhry, "Attacked and Hacked" and "The Gibraltar Hack: Anatomy of a Break-In", PC Week Labs, Oct. 1999, http://www.zdnet.com/eweek/stories/general/0,11011,2350743,00.html

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions", In Proc. 4th OSDI Symposium, San Diego, CA, Oct. 2000.

[12] A. Ghosh, T. O'Connor, G. McGraw, "An Automated Approach for Identifying Potential Vulnerabilities in Software", In Proc. 19th IEEE Symposium on Security and Privacy, Oakland, CA, May 1998, pp.104-114.

[13] S. Gribble, "A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction", Ph.D. thesis, UC Berkeley, Sep. 2000, p.6.

[14] I. Goldberg, D. Wagner, R. Thomas, E. Brewer, "A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker", In Proc. 6th USENIX Security Symposium, San Jose, CA, July 1996, pp.1-13.

[15] S. Hillier, "Inside Microsoft Visual Basic, Scripting Edition", Microsoft Press, Nov. 1996.

[16] P. Herrmann, H. Krumm, "Trust-Adapted Enforcement of Security Policies in Distributed Component-Structured Applications", In Proc. 6th IEEE Symposium on Computers and Communications, Hammamet, Tunesia, July 2001, pp.2-8.

[17] P. Iglio, F. Fraticelli, L. Giuri, "Rule-Based Filtering for Java Applets", In Proceedings of 14th Annual Computer Security Applications Conference, Phoenix, AZ, Dec. 1998, pp.112-119.

[18] B. Lampson, "Hints for Computer System Design", In Proceedings of 9th ACM Symposium on Operating Systems Principles, vol.17 (5), Bretton Woods, NH, Oct. 1983, pp.33-48.

[19] M. Monroe, "Security Tool Review: TCP Wrappers", ;login:, 18(6) 1993.

[20] F. Schneider, "Towards Fault-Tolerant and Secure Agentry", In Proc. 11th International Workshop on Distributed Algorithms, LNCS 1320, ACM-SIGPLAN, Springer-Verlag, Saarbruecken, Germany, Sept. 1997, pp.1-14.

[21] J. Smith, S. Doherty, O. Leahy, D. Tynan, "Protecting A Private Network: The AltaVista Firewall", Digital Technical Journal, Vol.9, No.2, Oct. 1997, pp.15-32.

[22] Security Focus, http://www.securityfocus.org/

[23] V. Swarup, "Automatic Generation of High Assurance Security Guard Filters", National Computer Security Conference, Washington D.C., Oct. 1994.

[24] Denial of Service Attacks and Buffer Overrun Vulnerabilities: in "Microsoft System Monitor ActiveX Exploitable Unicode Buffer Overflow Vulnerability", "Ultraseek 3.x Remote DoS Vulnerability", http://www.ussrback.com/

[25] D. Wagner, J. Foster, E. Brewer, A. Aiken, "A First Step Towards Automated Detection of Buffer Overflow Vulnerabilities", In Proc. Network and Distributed System Security Symposium, San Diego, CA, Feb. 2000, pp.3-17.

[26] "Extensible Markup Language (XML) 1.0 (Second Edition)", Oct. 2000, http://www.w3c.org/TR/REC-xml