Secure Migration of Legacy Applications to the Web

Zisis Karampaglis[1], Anakreon Mentis[1], Fotios Rafailidis[1(✉)], Paschalis Tsolakidis[2], and Apostolos Ampatzoglou[1]

[1] Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
{zkarampa,anakreon,frafaili,apamp}@csd.auth.gr
[2] Chalmers University of Technology, Gothenburg, Sweden
pastso@chalmers.student.se

Abstract. In software engineering, migration of an application is the process of moving the software from one execution platform to another. Nowadays, many desktop applications tend to migrate to the web or to the cloud. Desktop applications are not prepared to face the hostile environment of the web where applications frequently receive harmful data that attempt to exploit program vulnerabilities such as buffer overflows. We propose a migration process for desktop applications with a text-based user interface, which mitigates existing security concerns and enables the software to perform safely in the web without modifying its of the source code. Additionally, we describe an open source tool that facilitates our migration process.

Keywords: Software migration · Web application · User interface · Legacy application

1 Introduction

Migration of a software application is the process of moving it from one execution platform to another that is thought to be a more fitting one for the purpose of its operational use. For example, migration could mean replacing a Windows-based environment with a Linux-based environment or vice versa.

Nowadays, the growing trend for providing Software as a Service (SaaS) and the globalization of modern economy, demand the transfer of many systems to the web or even to the cloud. Such a transfer involves obvious implementation costs and security-related risks as well. Web applications are exposed to a large and distributed user base that, by accident or malice, can provide input that is harmful to the application execution and stored data.

In this paper we propose a migration process for *legacy* applications to the web, that protects the application from harmful user provided input. The proposed process uses an open-source tool developed by the authors for automatically adapting the text-based user interfaces to web-based UI without changing the program code.

A. Cerone et al. (Eds.): SEFM 2012 Satellite Events, LNCS 7991, pp. 229–243, 2014.
DOI: 10.1007/978-3-642-54338-8_19,

In Sect. 2, we discuss the current state of the art on how to modernize legacy applications and on data sanitization techniques. In Sect. 3, we describe the proposed migration process, along with a qualitative evaluation. Section 4 precuts the developed tool. Section 5 examines threats to validity while Sect. 6 provides conclusions derived from the outlined research and proposes future research and development prospect.

2 Related Work

In this section of the paper we provide background information and related work on the problems addressed in this paper, i.e. modernization of legacy applications and data sanitization. More specifically, in Sect. 2.1 we provide a literature review of studies on processes for migrating legacy applications to more modern environments. In Sect. 2.2, we describe basic concepts of data sanitization and related work on tools that perform these tasks.

2.1 Modernization of Legacy Application

Bringing legacy applications at par to the latest technological standards has been the focus of many research efforts. They propose a wide variety of processes and methods to modernize the software components of the legacy applications as well as the hardware they perform on. Modification of legacy code is an expensive and error-prone task which most research efforts address only partially. A middleware is usually introduced to provide the necessary stepping stone for the migration of the application into a modern execution platform. Software modernization is achieved by enhancing application properties such as availability, usability, security, flexibility, interoperability, expandability and maintainability.

Modern operating systems prevent or limit direct access to system components that raise interoperability issues with some legacy applications. In [19], the authors propose the use of a Virtual Desktop Infrastructure that executes the legacy application in its own OS. In [12], the authors propose the use of a POSIX shell interpreter and an applet to allow the execution of an application from different operating systems. In our approach we achieve migration of desktop applications to the Web by introducing a TUI middleware maintaining platform independency. Similarities with our approach can be found in [13], where the authors describe two problems that legacy applications must surpass to achieve web-integration: platform dependencies and quality requirements that the application should meet. The authors show that both problems are efficiently resolved by a middleware product.

In [10], several case studies of test-driven methods are presented to define the flexibility of a legacy application in terms of security, by determining the degree of coupling between business logic and access control. Highly coupled applications are less flexible and require more modifications in the implementation to enhance security. In [9], the author presents an Interface Adapter Layer to enable communication between a Common Object Request Broker Architecture

(CORBA) platform and a legacy application. With this technique, the legacy application maintains its autonomy, enhances its availability and improves its security. The TUI library in this paper operates as a communication bus between the Web and the legacy application, obtaining the aforementioned benefits.

Approaches to increase flexibility appear in [18] where the authors propose a framework to enable Quality of Service (QoS) based on a network layer mechanism, for legacy applications. The main constraint of introducing such a framework is the inflexibility to modification and recompilation that comes with legacy code. In order to surpass this obstacle the authors use a middleware component that bridges the gap between the application and the network QoS entities. The same applies for the Web implementation of our approach, which needs only the recompilation of the UI Library and not of the legacy code.

An approach to usability comes from [7] where the authors suggest two proof-of-concept techniques to change the user interface of legacy applications into an innovative one. They separated the graphic output subsystem and the user interface from the application and modified them to support modern ways of interaction without changing the core of the application. In this paper the API/TUI is used in the same way to allow interaction from the Web without altering the code of the legacy application.

An upward trend exists that supports the transition of legacy applications towards a Service Oriented Architecture (SOA). Legacy applications are usually transformed into Grid services or services of cloud computing. In [1], the authors propose three approaches to transition legacy applications to Web Services through Service Oriented Architecture (SOA). The reason for this transition is the increased need for new features in organizations legacy IT infrastructure. The three approaches are: a session based approach that changes only the Graphical User Interface (GUI) keeping the code of the legacy application unmodified, a transaction based approach to cover security issues that come with the transition and a data based approach that expose the data of the application to the Web.

In [21] the authors propose a method for migrating legacy applications into Grid Services. The otherwise standalone applications are reused and wrapped into the Grid technology by using the Globus Toolkit. The main benefit of this approach is that multi-user access can be achieved without modifying the code of the application for Internet functionality. Similar to [21], the authors of [6,8,11] present their proposals for migrating legacy code into Grid services without changing the source code of the application. The architecture of such an approach is identical to the architecture of our tool. In [20], the authors provide a solution to migrate a legacy application to the cloud. The Application Migration Solution (AMS) reconstructs the GUI of the application without changing its source code. The solution also supports the compilation of more than one application together, in order to create a more powerful, in terms of features, application.

The authors of [4] use the Ubiquitous Web Application (UWA) framework to reengineer a legacy application into the web. The benefits of their approach

are good documentation and high usability/maintainability for the reengineered application as opposed to a production-oriented approach based on studies from past years. An extension of the above work from the same authors is in [5]. They used the Ubiquitous Web Applications Design Framework (UWA) and an extended version with Transaction Design Model (UWAT+). Through this approach they leveraged legacy applications into rich Internet ones and due to the formality of the framework they minimized the time needed for the transition.

Expandability towards modern technologies seems that is not a top priority for legacy applications. The authors of [2] present a solution to the problem of continuous re-engineering for legacy applications. With constant maintenance to keep an application up to date with the latest technological standards, its architectural structure becomes more difficult to expand. To mitigate the problem they suggest the use of ArchJava in order to ensure the integrity of the original architectural structure.

2.2 Data Sanitization

Sanitization takes user input and transforms it in order to eliminate potential threats for the data and operation of the application. Threats exist due to the use of characters known as *metacharacters* that have special meaning, when processed by the various parts of a system. The transformation of user input is applied by removing characters based on two methods. These methods are distinguished by using a white or black list of characters, which contain characters that respectively can or cannot be harmful for the application.

For example, an SQL Injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application using meta-characters or special characters of SQL, such as ', ', AND, =. An SQL injection attack can read sensitive data, modify data, and execute administration operations. The example from [15] in Listing 19.1 uses a *Prepared Statement*. Consider the string *john* is the value for input parameter *user_name*, which is a valid input. The application will search a user with the *user_name john* and will return all information for this user. On the opposite, if someone passes the string *1 OR 1=1*, this leads to an attack because the query that the program submits to the database contains a tautology in the WHERE clause and gain access to sensitive information regarding database users.

Listing 19.1. Sample code for SQL Injection

```
String firstname = req.getParameter("firstname");
String query = "SELECT * FROM authors
       WHERE firstname = ? ";
PreparedStatement pstmt =
       connection.prepareStatement( query );
pstmt.setString( 1, firstname);
ResultSet results = pstmt.execute( );
```

To prevent an SQL Injection Attack, we apply data sanitization. The quotes ', ' and = are removed from the string and the sanitized string will be *1OR11*,

which cannot be a tautology. Several characters can be meta-characters depending on the language of the application. Data sanitization aims to prevent harmful input data to be inserted in an application. In [14] the authors propose a technique of automatic query sanitization to prevent SQL Injection attacks. They use a combination of static analysis and program transformation to automatically instrument web applications with sanitization code.

There are three different ways to apply data sanitization to the user data. The first way is to use tools, which take user data, sanitize them and produce as output the sanitized user data to be sent to the web application. *Urlrewritefilter* is based on a tool that rewrites url using certain filters and sends the modified url to the web application. Another tool for sanitizing html tags, attributes and values is *jsoop*. *Jetscripts* Data Sanitizer and XSS cleaner prevents SQL Injection and XSS attacks by cleaning or sanitizing user-submitted data. This tool is intended for users who write or modify scripts, or want an extra measure of protection against malicious users. It requires some knowledge in php scripting. The sanitizer can work in various modes such as numeric only, alphabetic only, alphanumeric only, alphanumeric with punctuation and email validation mode. Additionally to the above modes common command entities and Javascript specific entities are removed.

The second way is to use libraries that offer functions which sanitize input data. The ESAPI library by OWASP provides libraries for many programming languages such as Java, .net, ASP, PHP, PHP, ColdFusion, Python, JavaScript, Objective-C, Ruby, C, CPP and Perl.

The third way is to use embedded functions in various languages that sanitize input data depending on special characters for each language. At php, function *mysql_real_escape_string()* sanitizes special character of mysql at a string, which is proposed to be sent to the mysql database. Other functions are *filter_input()*, *escapeshellarg()* and *urlencode* for php, etc. Finally, tools that are used to statically analyze the vulnerabilities of code written in C are presented and compared in [3].

Security issues caused by invalid sanitization of user-provided input is the focus of [16]. Cross-site scripting (XSS) exploits is prevented without any modifications to the application implementation.

3 Migration Process

A typical architecture for desktop applications is shown in Fig. 1. The application receives input from the keyboard and renders the output on the user's screen. Both input and output are handled by a TUI library responsible for the translation of user input into a series of events and the application output into user interface elements such as windows, labels and buttons. The application interaction with the user consists of appropriate reactions to the events generated by the TUI library.

On the other hand, in a web–based architecture the user–provided data are encoded as POST or GET variables transferred via the HTTP protocol. An

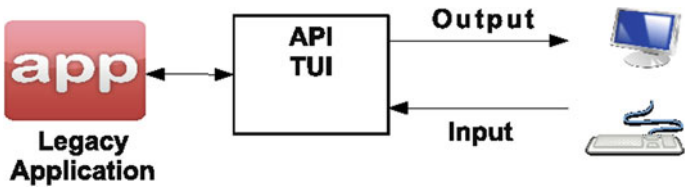

Fig. 1. Legacy application architecture

application residing on the server processes the received user input and responds with an HTML document displayed to the user by a web browser. To compensate for the stateless nature of the HTTP protocol, web applications store information that are required in conversations between the client and the server in cookies or include the values in POST or GET variables in every exchange of data. Clearly, this is in contrast with desktop applications that can preserve state between subsequent user interactions.

Desktop applications operate in a relatively secure environment where the user–provided input can be trusted and is checked only for accidental mistakes. In contrast, malformed user input is often deliberately sent to web applications with the explicit goal to disable it's normal operation or to gain access to sensitive data managed by the considered web application.

The proposed migration process aims at (a) transferring the application to the web without modifying the application code base and (b) secure the application from harmful input received from the web interface.

We propose a two step process to achieve those goals. To fulfill the first goal we replace the TUI library with a modified version that does not receive the user–provided input from the keyboard; instead it uses the information provided by the GET and POST variables. Moreover, the modified TUI library produces an HTML document which, when rendered by the browser, displays the application output and TUI elements that enable further interactions. The second goal is realized by the sanitizer component that blocks malformed input or transforms potentially dangerous input into harmless data. The application architecture produced from the proposed method is shown in Fig. 2.

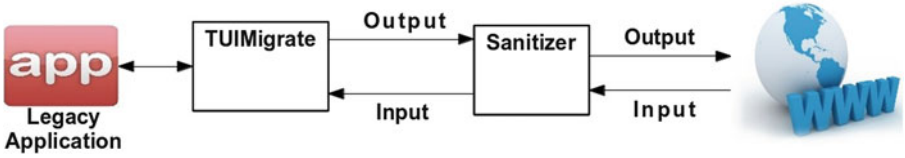

Fig. 2. The architecture of the web–enabled legacy application

The proposed process clearly addresses both migration goals. The modified TUI library retains the same API with the original version. All that is required

for enabling the application in the web is a recompilation with the modified TUI library. Also, since the sanitizer is not part of the application it can be modified independently. We could integrate one of the many available open source tools that provide sanitization services (see Sect. 2.2 for an overview).

In order to validate the usefulness of our approach, we have conducted an interview with the project manager of a CRM created in the early 1980s for the DOS operating system. After three years of development, the CRM was migrated to the Windows OS. Currently, they are considering migrating the application to the web which they estimate it will require an additional five year period. The duration of migrating from Windows to the Web is estimated to be as long as the transition from DOS to Windows, because of increased security threats in the execution environment of web applications.

The above case suggests that an automated process for a secure migration of legacy applications to the web would be useful for the software industry.

4 The Modified UI Library

In this section we show in detail the required process for modifying a Text User Interfaces (TUI) library in order to enable applications that use it in the web. The process is applied on an open source TUI named Turbo Vision [17], a framework developed by Borland and later placed in the public domain. It provides a reach user interface with various components such as menus, check boxes, buttons, and many others. Figure 3 shows a Turbo Vision application.

The modified library expects user input not from input devices such as the keyboard or the mouse but from a file or the standard input. Also, the application output is not rendered on the screen. Instead, it is transformed into an XML document which can be rendered to the user's browser with the help of style sheets or some other method able to produce an HTML document. Our library is currently in beta version and is distributed[1] freely under a permissive free software license.

TVision components correspond to C++ classes that form the hierarchy shown in Fig. 4. TView is the base class inherited by all components. We declared additional methods in this class responsible for exporting the component's state into XML and for recovering a previous state from a textual format. When needed, derived classes modify accordingly the implementation of the added methods.

TGroup represents compound components that consist of other components. In this class, the added methods of TView are reimplemented to invoke the respective method for each of the contained components. For example, the method responsible for serializing the component's state into XML invokes the same method of each contained component and returns the merged obtained output.

TVision applications consists of one or more TDialog instances where only one of them is active at any time. Dialogs are modal, the user can interact only

[1] http://sourceforge.net/projects/tuimigrate

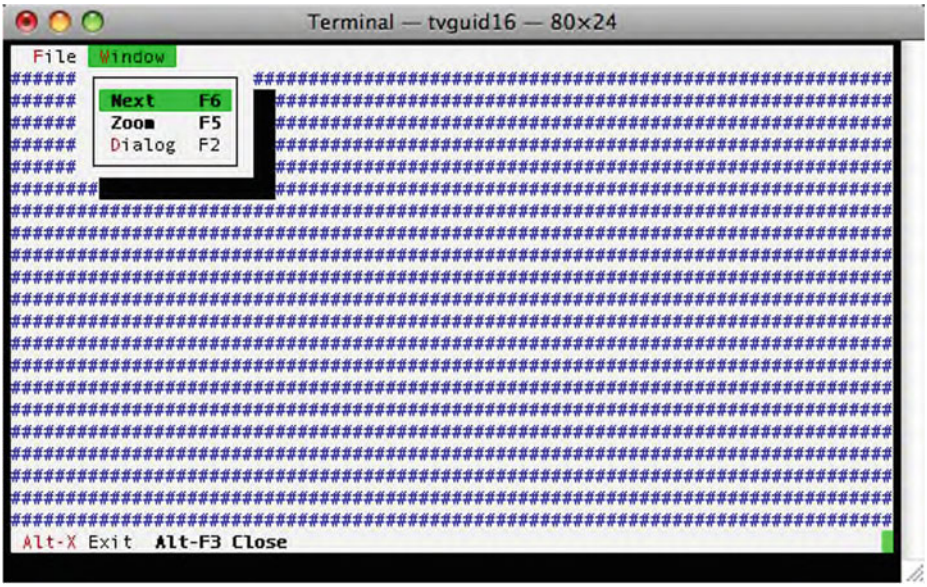

Fig. 3. An example TUI application

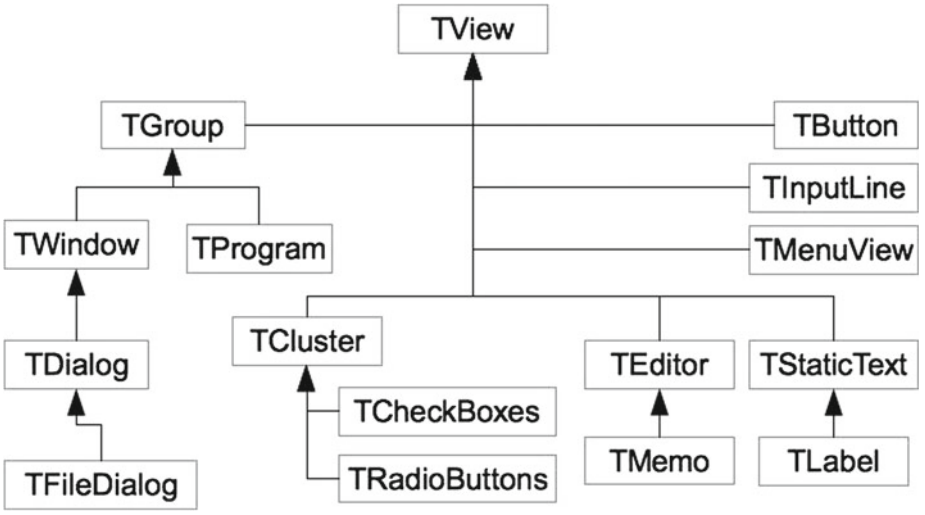

Fig. 4. TVision classes and their hierarchy

with the active dialog and the other dialogs of the application are inaccessible until the active one is closed. TDialog is a container for other components shown in Table 1.

Table 1. Main TVision components

Component	Description
TButton	Buttons
TCheckBoxes	Group of check-box components
TFileDialog	File selector
TInputLine	Input for one line of text
TLabel	Descriptive labels for other components
TMemo	Multi-line text input control
TMenuView	Abstract base class for menus
TRadioButtons	Group of radio button components

User input from the keyboard or interactions with the mouse are translated into events which the application can bind and react accordingly. In our modified version of the library we retain the same mechanism only that the events are no longer produced from input devices but from a specially formatted text content that describes user actions. In essence, our modified implementation emulates the user interactions and as far as the application code is concerned, nothing is changed.

4.1 Input/Output

The library binds a port in order to communicate with the client application (e.g. the sanitizer) and accepts two commands. One command produces an XML description of the currently accessible dialog and the components it contains (e.g. forms, menus, buttons etc.), while the other command asks the library to produce interaction events from a textual representation. In the expected work-flow, the user is presented with an HTML form produced by the XML description of the active dialog, his interactions with the form are reflected on the values received by the web server and the sanitizer submits to the library a description of the user actions (e.g. the button clicked, value of a text field). Those actions are translated into events and the new updated state of the application is presented to the user with an other form.

The XML content that describes the application User Interface consists of three main parts that list the attributes of application menus, windows and dialogs:

Menu. Menus contain menu items as well as nested menus. Menu items can be identified by a name or by an identifier and the schema defines the two respective attributes. Moreover, menu items can be associated with an action performed when the user selects it.

Window. This part describes the windows of the application. Windows have a title and contain other components.

Dialog. The document contains a dialog description only if there is an active dialog and describes all the components contained in the dialog. Since dialogs

are modal, the previous two sections of the document are not included in order to prevent the user from accessing menu entries or windows.

All elements (windows, dialogs, menus and components) have the following common attributes: a unique identifier, a type, a content attribute that stores the component's value and a name. In Listing 19.2 we show an excerpt of the XML Schema that defines in detail the structure of the XML document produced by the library.

Listing 19.2. A small part of the output XML schema

```
<xs:element name="dialog">
    <xs:complexType>
        <xs:choice maxOccurs="unbounded">
            <xs:element ref="label"/>
            <xs:element ref="text"/>
            <xs:element ref="button"/>
            <xs:element ref="checkbox"/>
            <xs:element ref="radiobutton"/>
        </xs:choice>
    </xs:complexType>
    <xs:key name="formItemKey">
        <xs:selector xpath="*"/>
        <xs:field xpath="@id"/>
    </xs:key>
    <xs:keyref name="formItemRef" refer="formItemKey">
        <xs:selector xpath="label"/>
        <xs:field xpath="@idref"/>
    </xs:keyref>
</xs:element>
<xs:element name="checkbox">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="item">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:string">
              <xs:attribute name="selected"
              type="xs:boolean"/>
              <xs:attribute name="id"
              type="xs:int" use="required"/>
     ....
    </xs:sequence>
    <xs:attribute ref="id" use="required"/>
  </xs:complexType>
</xs:element>
```

User actions are provided in a textual description where each line corresponds to an action. Lines are expected to have one of the forms shown in Table 2.

Table 2. Available instructions for altering the TUI state

Form	Description
name :: value	Assigns a value to the component identified by the given name.
id <: value	Assigns a value to the component identified by the given id.
name :> index	Activates a component with the given index contained in a component identified by the provided name.
id <> index	Has the same effect as the previous command for the container with the provided unique identifier.

Let us examine a user session to demonstrate our approach and see how user interactions are handled by the application with the help of the modified TUI library. At first the user is presented with an HTML form produced by the window shown in Fig. 3 and selects the "Dialog" menu. The form is submitted to the web server and the sanitizer components delivers to the application the following instruction:

```
m202 <: active
```

which selects the application's "Dialog" menu and performs the associated action. The performed action creates a new modal dialog shown in Fig. 5 described in XML in Listing 19.3. After the user modifies the values of the various controls, it submits the form with the "OK" button delivering to the application the following action description:

```
1 Delivery Instructions : : Hurry!
2 Cheeses :> 0
3 8 <> 2
4 6 <: Runny
5 OK : : true
```

In the first line, the library is instructed to set the value of the text component associated with the label named "Delivery Instructions" to "Hurry!". In Listing 19.3, the "idref" attribute of the label shows which component is associated with the label. In the second line, the library finds the check box group component associated with the label "Cheeses" and selects the first check box (index zero). The third line performs the same action but it refers to the check box group by id and selects the third check box. As a result of lines two and three, "Tilset" is the only unselected check box. The fourth instruction sets the value of the component with id 6 to "Runny" effectively selecting the second radio button of the radio group labeled "Consistency". Finally, the last action selects the button labeled "OK" and performs its associated actions.

Listing 19.3. XML produced from TUI of figure 5

```
<?xml version="1.0" encoding="UTF-8"?>
<program xmlns:xsi=
       "http://www.w3.org/2001/XMLSchema-instance">
  <dialog>
    <button id="1">Cancel</button>
    <button id="2">OK</button>
    <label id="3" idref="4">Delivery Instructions</label>
    <text id="4"><![CDATA[Phone Mum!]]></text>
    <label id="5" idref="6">Consistency</label>
    <radiobutton id="6">
        <item selected="false" id="0">Solid</item>
        <item selected="false" id="1">Runny</item>
        <item selected="true" id="2">Melted</item>
    </radiobutton>
    <label id="7" idref="8">Cheeses</label>
    <checkbox id="8">
        <item selected="true" id="0">Hvarti</item>
        <item selected="false" id="1">Tilset</item>
        <item selected="false" id="2">Jarlsberg</item>
    </checkbox>
  </dialog>
</program>
```

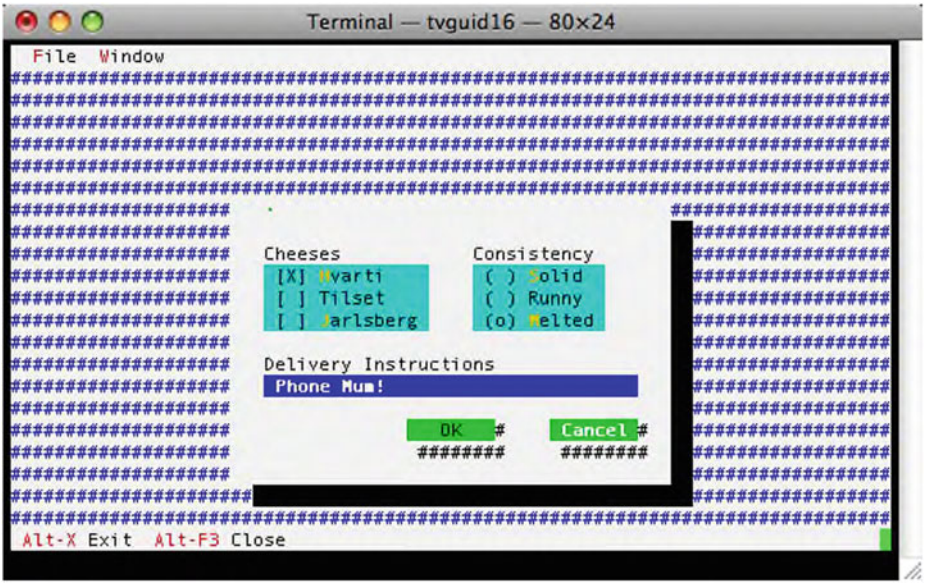

Fig. 5. An example of a modal dialog

5 Future Work

At the current preliminary stage of development, we have provided an alternative implementation of a TUI library API that enables legacy applications to operate in a Web environment. We are currently extending the sanitization component with functionality offered by the OWASP security library which detects and blocks user input that cause SQL injection attacks. When all components of the architecture of the proposed migration approach are completed, we could demonstrate a more detailed example with a web interface and the communication between the interface and the application. Then, we will be able to validate empirically whether TUI application users are equally satisfied by the automatically generated web-based UI. In addition, we will use the Turbo Vision library for more applications and try to apply the above method to other TUI libraries.

6 Conclusions

This work proposes a process for automatically migrating legacy applications to the web. It is a general method in the sense that it places no constraints on the design and implementation of a legacy TUI application. As far as the application is concerned, nothing is changed. It still invokes the same methods provided by the TUI API although the user input is now provided by data sent remotely with HTTP requests and the produced output is now transformed into HTML instead of being displayed on the user's screen. Also, the received data is cleared of special characters that may threaten the integrity of the application data or the security of the application. Common cases of errors such as SQL injection attacks are handled by the data sanitizer. In order to demonstrate the feasibility of the approach, we have implemented the API of a TUI library and a tool that automatically transforms TUIs (textual user interfaces) to HTML based UIs. The proposed process, reduces the migration cost and enables secure access to previously locally executed applications in the demanding web environment.

Acknowledgement. This work was performed in the framework of the TRACER (09SYN-72–942) project, which is funded by the Cooperation Programme of the Hellenic Secretariat for Research & Technology.

References

1. Al Belushi, W., Baghdadi, Y.: An approach to wrap legacy applications into web services. In: 2007 International Conference Service Systems and Service Management, pp. 1–6 (2007)
2. Abi-Antoun, M., Coelho, W.: A case study in incremental architecture-based reengineering of a legacy application. In: 5th Working IEEE/IFIP Conference on Software Architecture, 2005, WICSA 2005, p.p. 159–168 (2005)

3. Chatzieleftheriou, G., Katsaros, P.: Test driving static analysis tools in search of C code vulnerabilities. In: Proceedings of the 35th IEEE Computer Software and Applications Conference Workshops (COMPSACW), Munich, Germany, pp. 96–103. IEEE Computer Society (2011)
4. Distante, D., Perrone, V., Bochicchio, M.A.: Migrating to the Web legacy application: the Sinfor project. In: Proceedings of the Fourth International Workshop on Web Site Evolution, 2002, pp. 85–88 (2002)
5. Distante, D., Tilley, S., Canfora, G.: Towards a holistic approach to redesigning legacy applications for the Web with UWAT+. In: Proceedings of the 10th European Conference on Software Maintenance and Reengineering, 2006, CSMR 2006, pp. 5–10 (2006)
6. Lu, F., Huang, H., Xu, Z., Yu, H.: A middleware for legacy application wrapper. In: First International Conference on Semantics, Knowledge and Grid, 2005, SKG '05, pp. 47 (2005)
7. Besacier, G., Vernier, F.: Toward user interface virtualization: legacy applications and innovative interaction systems. In: EICS '09: Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 57–166. New York (2009)
8. Kacsuk, P., Goyeneche, A., Delaitre, T., Kiss, T., Farkas, Z., Boczko, T.: High-level grid application environment to use legacy codes as OGSA grid services. In: GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, pp. 428–435. Washington (2004)
9. Konstantas, D.: Migration of legacy applications to a CORBA platform: a case study. In: Proceedings of the IFIP/IEEE International Conference on Distributed Platforms: Client/Server and Beyond: DCE, CORBA, ODsanitP and Advanced Distributed Applications, pp. 100–112 (1996)
10. Le Traon, Y., Mouelhi, T., Pretschner, A., Baudry, B.: Test-driven assessment of access control in legacy applications. In: 2008 1st International Conference on Software Testing, Verification, and Validation, pp. 238–247 (2008)
11. Zhu, L., Matsunaga, A., Sanjeepan, V., Lam, H., Fortes, J.A.B.: Application modeling and representation for automatic grid-enabling of legacy applications. In: First International Conference on e-Science and Grid Computing, pp. 8–31 (2005)
12. Marosi, A.C., Balaton, Z., Kacsuk, P.: GenWrapper: a generic wrapper for running legacy applications on desktop grids. In: IEEE International Symposium on Parallel & Distributed Processing, 2009, IPDPS 2009, pp. 1–6 (2009)
13. Mondal, S.A., Gupta, K.D.: Choosing a middleware for web-integration of a legacy application. SIGSOFT Softw. Eng. Notes **25**(3), 50–53 (2000). (New York)
14. Mui, R., Frankl, P.: Preventing SQL injection through automatic query sanitization with ASSIST. In: Fourth International Workshop on Testing, Analysis and Verification of Web Software, EPTCS 35, Antwerp, pp. 27–38 (2010)
15. Owasp. https://www.owasp.org/
16. Saxena, P., Molnar, D., Livshits, B.: SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In: CCS '11: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 601–614. New York (2011)
17. Sigala Turbo Vision. http://www.sigala.it/sergio/tvision/index.html
18. Tsetsekas, C., Maniatis, S., Venieris, I.S.: Supporting QoS for legacy applications. In: Lorenz, P. (ed.) ICN 2001. LNCS, vol. 2094, pp. 108–116. Springer, Heidelberg (2001)

19. Wong, D.: Kickin' it old school!: dealing with legacy applications. In: SIGUCCS '08: Proceedings of the 36th Annual ACM SIGUCCS Fall Conference: Moving Mountains, Blazing Trails, pp. 55–58. New York (2008)
20. Meng, X., Shi, J., Liu, X., Liu, H., Wang, L.: Legacy application migration to cloud. In: 2011 IEEE International Conference on Cloud Computing (CLOUD), pp. 750–751 (2011)
21. Xiong, Y., Su, D.: Wrapping legacy applications into grid services: a case study of a three services approach. In: Shen, W., Luo, J., Lin, Z., Barthès, J.-P.A., Hao, Q. (eds.) CSCWD. LNCS, vol. 4402, pp. 520–529. Springer, Heidelberg (2007)