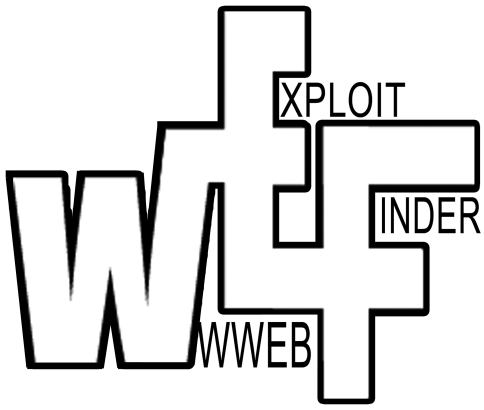


WEF – Web Exploit Finder

Dokumentation zum Praktikum Softwaretechnik

SS 2006

Medieninformatik

Hochschule der Medien, Stuttgart

Thomas Müller (tm026)

Benjamin Mack (bm022)

Mehmet Arziman (ma018)

Betreut von:

Prof. Dr. Roland Schmitz

Inhaltsverzeichnis

1 Motivation

In regelmäßigen Abständen kann man über neue Sicherheitslücken im Microsoft's Internet Explorer oder in Mozilla's Firefox lesen. Einige dieser Lücken eignen sich um den Browsern schadhaften Quellcode unterzujubeln. Diese Angriffe werden auch als *Remote Code Execution*-Attack bezeichnet und basieren in der Regel auf Buffer-Overflows (Speicherüberlauf) in den Applikationen.

Diese Schwachstellen treten natürlich nicht nur bei Browsern auf, sondern bei fast allen Diensten und Applikationen, die Teil des Internets sind oder dieses als Kommunikationsplatform nutzen.

Warum also der Fokus auf Internet-Browser?

Internet Browser haben zwei entscheidende Problematiken:

Erstens stellen sie die Schnittstelle zum WorldWideWeb dar indem die Rendering-Engine Hypertext und andere Webinhaltefür den Menschen sichtbar macht. Dies hat zur Folge, dass alle Teile einer Internetseite vom Browser interpretiert und weiterverarbeitet werden müssen, was eine komplexe und fehleranfällige Architektur zur Folge hat, vor allem was mobilen Code (JavaScript, Java, ActiveX, XUL etc.) betrifft.

Zweitens ist der Browser wohl das am häufigsten verwendete Programm aus der Familie der potentiell gefährdeten Software. Im Gegensatz zu Server-Software wird der Browser von technisch nicht versierten Benutzern verwendet, denen oft weder die Risiken noch mögliche Gegenmaßnahmen bekannt sind. Und selbst versierte Nutzer setzen sich oft dem Risiko eines Angriffs aus.

Unser Ziel war es deshalb, ein System zu entwickeln, das auf Abruf schadhafte Internetseiten zu identifizieren kann.

Zusätzlich kann das Projekt auch als Framework dienen um andere Security/Sandbox-Tests für Windows-Applikationen in einer gesicherten und leicht wartbaren virtualisierten Umgebung durchzuführen.

2 Beschreibung der Idee

Zu Beginn des Projektes waren zunächst einige wichtige Fragen zu klären.

- Wie soll der Begriff **„schadhaft"** für unser Projekt definiert sein?
- Welche Möglichkeiten gibt es schadhafte Internet Inhalte zu erkennen?
- Wie muss ein entsprechendes Software-System konzipiert sein?

Als **„schadhaft"** versteht unser Projekt Webseiten, welche Schadsoftware

(Viren, Würmer, Trojanische Pferde, Keylogger, ...) auf einem Client-Rechner installieren und ausführen. Wir konzentrieren uns auf Schadsoftware, die sich ohne Benutzerinteraktion installiert und somit sogar für versierte Benutzer nicht erkennbar ist.

Diese Einschränkung wurde bewusst gewählt um nur Seiten zu finden, welche tatsächlich Sicherheitslücken im Browser ausnutzen. Seiten, die den Benutzer überlisten oder Seiten, die modifizierte Software zum Download anbieten, sind nicht Ziel unserer Suche. Es ist aber im Konzept berücksichtigt, diese Funktion zu einem späteren Zeitpunkt nachzurüsten.

Um die Frage zu beantworten zu können, wie man schadhafte Seiten erkennen kann, haben wir zunächst zusammengestellt, was die Ziele des Angreifers sind und wie er diese Ziele erreicht:

Ein Angreifer will den Rechner dauerhaft kompromittieren und muss hierfür den aktuellen Zustand des PCs verändern. Ein mögliches Szenario wäre:

1. Der Angreifer bringt mit Hilfe eines Bufferoverflows im Browser seinen eigenen Code (Shellcode) zur Ausführung.
2. Da die Funktionalität durch die geringe mögliche Größe stark eingeschränkt ist, wird in der Regel nur versucht, weiteren ausführbaren Code aus dem Internet herunterzuladen und auf dem Rechner zu starten.

Diese Mini-Applikation wird häufig als „Dropper" oder „Downloader" bezeichnet, da sie die eigentliche Malware auf den Rechner lädt und diese mittels Registry-Einträgen bei folgenden Systemstarts zur Ausführung bringt.

Um derartige Veränderungen am Zustand des Rechners zu erkennen gibt es zwei unterschiedliche Möglichkeiten.

1. *Intrusion Detection: Vor und nach dem Besuch einer verdächtigen Internetseite den Zustand des Sandbox Rechners zu ermitteln und die beiden Ergebnisse zu vergleichen.*
   Zur Ermittelung des Zustandes kann ein Auflisten aller relevanten Dateien und Registry-Einträge in Verbindung mit Prüfsummen über diese Daten verwendet werden. Dadurch können neu erzeugte sowie veränderte Dateien erkannt werden. Dieses Verfahren wird von einem Projekt namens „HoneyClient" verwendet. Die entscheidenden Nachteile dabei sind aber die große zeitliche Verzögerung der Entdeckung sowie die schlechte Performance.

2. *Rootkit: Erkennung mittels Modifikation des Betriebssystems.*
   Hier werden die relevanten Systemcalls überwacht und gegebenenfalls protokolliert. Eine derartige Modifikation des Betriebssystems ist aber standardmäßig nicht vorgesehen und benötigt einen tiefen Eingriff in den Kernel. Diese Technik findet häufig auch in sogenannten *Rootkits*

Verwendung und wird deshalb meist selbst als Rootkit bezeichnet.

Wir haben uns aufgrund der erwähnten Performance-Vorteile für die Rootkit-Lösung entschieden.

Um ein solches System später auch für Forschungszwecke sinnvoll einsetzen zu können, muss es folgenden Anforderungen genügen:

- Es sollte vollständig autonom arbeiten um so gut wie keine Benutzerinteraktion benötigen.
- Es sollte sich remote aus der Ferne überwachen und steuern lassen, z.B. mit Hilfe eines Webinterfaces.
- Es sollte skalierbar und erweiterbar sein.
- Komponenten für die eigene Sicherheit des Monitoring Systems verhindern, dass es nicht selbst durch schadhafte Seiten infiziert wird.

3 Einschränkungen

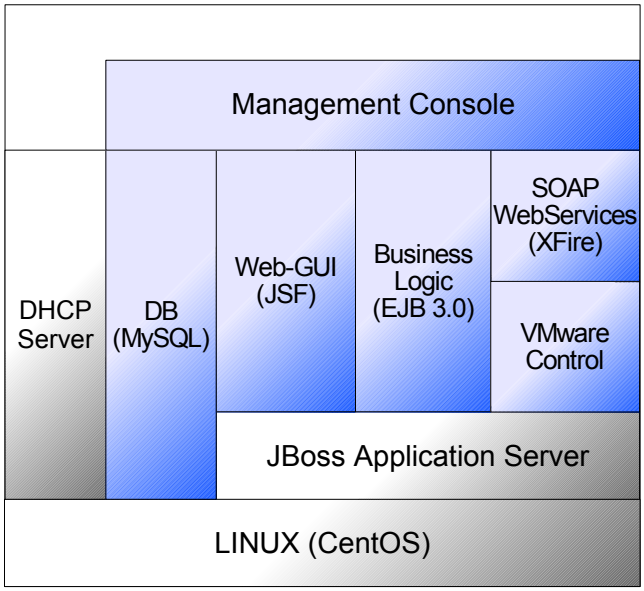
Bei der Erkennung von schadhaften Seiten haben wir uns zunächst für folgende Konfiguration des Testsystems entschieden:

- Windows XP Professional ohne Service Pack
- Keine Sicherheitsupdates installiert
- Ausführung unter dem Administrator-Account
- Verwendung des MS Internet Explorers 6
- Scripting und Java sind aktiviert

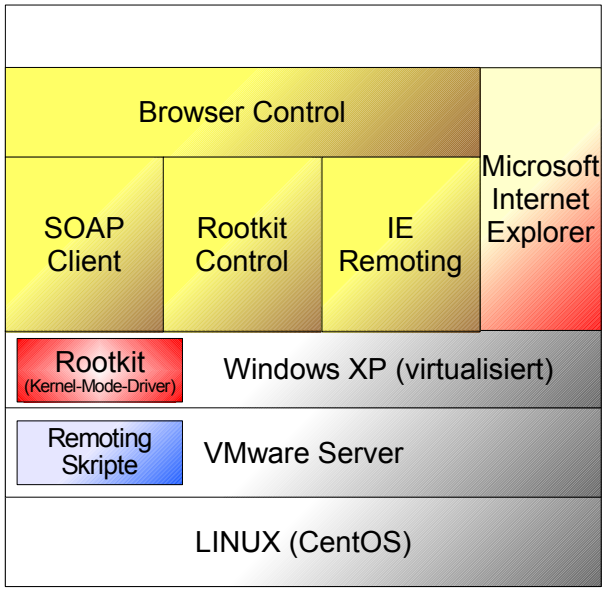
Diese Konfiguration wurde gewählt um eine möglichst große Angriffsfläche zu bieten. Sie simuliert den *Worst-Case,* der sich aber durchaus in der Konfiguration einiger Nutzer widerspiegelt.

4 Die System-Architektur

Übersicht



Workstation 1: Steuerung



Workstation 2: Sandbox

Kurzbeschreibung

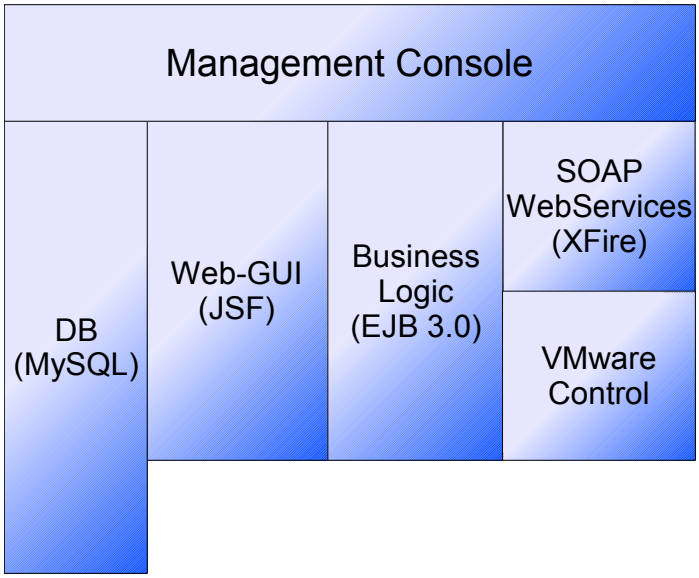
Um den Anforderungen aus Kapitel 2 gerecht zu werden, benötigt unsere System-Architektur folgende, noch abstrakt gehaltene Komponenten:

- Zum Schutz der Architektur und um parallel mehrere Seiten zu prüfen eine **Virtualisierungsschicht**:den VMware Server.

- Zur Erkennung der schadhaften Seiten die Betriebssystem-Modifikation: unser **Rootkit**
- Zur Steuerung des Rootkits und des Internet Explorers eine **Client seitige automatisierungs Komponente**: das Browser Control
- Und zur Steuerung und Konfiguration des ganzen Systems: Die Management Konsole

4.1 Management-Konsole

Übersicht



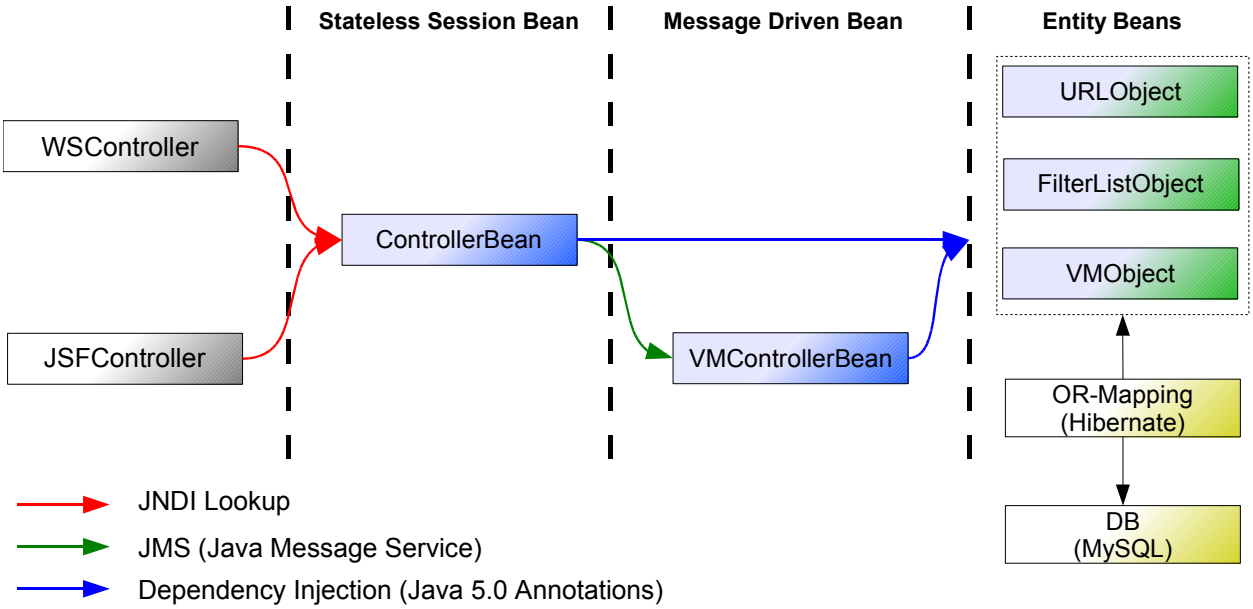
Kurzbeschreibung

Die Management-Konsole ist einer der Hauptkomponenten des Systems. Sie dient der Steuerung und Überwachung der virtuellen Windows-Instanzen. Sie besteht aus fünf Modulen.

- Das **VMware-Control-Modul** bietet eine Schnittstelle zur VMware Server Software auf der Sandbox Maschine.
- Die **SOAP-Webservices** werden für die Kommunikation zwischen Browser-Control auf der Sandbox Maschine und der Management-Konsole benötigt.
- Die **Business-Logik** enthält den logischen Ablauf des Systems und verwaltet das Datenmodells.
- Über das **Web-GUI** kann der Benutzer das System steuern und die aktuellen Vorgänge und Ergebnisse abfragen.
- In der **Datenbank** werden Konfigurationen die Ergebnisdaten gespeichert.

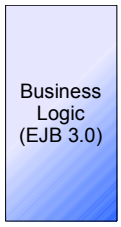
4.1.1 Schnittstellen der Management-Konsole

Übersicht



4.1.2 Business-Logik und Datenmodell (EJB 3.0)

Kurzbeschreibung der Komponente



Die Business-Logik ist die Kernkomponente des Systems. Sie enthält die Programm-Logik, verwaltet die Datenobjekte und überwacht die Zustände der weiteren Komponenten.

Eingesetzte Technologien

Für die Umsetzung dieser Komponente wurden folgende Technologien verwendet:

- Enterprise Java Beans (EJB) in der Version 3.0
- JBoss Application Server mit EJB3.0 Modul
- Hibernate als O/R-Mapper
- Eclipse WTP (Web Tools Plattform) + JBoss EJB3.0 Wizard

Verlauf der Entwicklung

Nachdem wir uns innerhalb des Teams auf die Notwendigkeit einer solchen Management-Konsole geeinigt hatten, beschäftigten wir uns mit der

technische Umsetzung. Es musste eine Entscheidung zwischen zwei möglichen Umsetzungen getroffen werden

- Realisierung als Windows-Applikation z.B. mit Java-Swing oder auf Basis von C# und .net
- Realisierung als Web-Applikation (PHP, Ruby on Rails, J2EE)

Da wir eine einerseits Plattform-Unabhängigkeit und zum anderen die das System von einem beliebigen Rechner aus steuern wollten, entschieden wir uns für die Web-Applikation.

Zunächst favirisierten wir eine Lösung mittels PHP, da auf diesem Gebiet bereits einiges an Erfahrung vorhanden war. Aufgrund der umfangreichen Anforderungen an die Schnittstellen und wegen der einfachen Erweiterung beschäftigten wir uns zusätzlich noch mit J2EE . Wir haben uns schliesslich für eine Implementierung mittels EJB 3.0 entschieden, da es einen deutlich einfacherer Einstieg bietet als die Vorgänger-Version.

Die Business-Logik wurde wie von EJB vorgesehen mittels Session Beans realisiert. Da die Business-Logik zum jetzigen Zeitpunkt noch überschaubar ist, verwenden wir aktuell nur eine einzige Stateless Session Bean (*ControllerBean.java*), die mit Ausnahme der asynchronen Funktionen die komplette Logik abbildet. Da wir innerhalb dieser Komponente keine Transaktionen ausführenund keinen Status benötigen verwenden wir die schnellere und ressourcensparende *stateless* Variante.

Nachdem die Grundfunktionalität implementiert war und uns noch genug Zeit für Ergänzungen blieb, haben wir uns dafür entschieden unsere Datenobjekte persistent in einer Datenbank abzulegen:

- *URLObject.java*: Repräsentiert eine einzelne URL und deren für das System relevanten Eigenschaften.
- *FilterListObject.java*: Repräsentiert eine Zeile in der Filterliste. Diese Filterliste dient der Konfiguration des Rootkits.
- *VMObject.java*: Repräsentiert eine virtuelle Windows XP Instanz und spiegelt den aktuellen Zustand dieser wieder.

Diese Objekte waren bereits als normale Java Objekte vorhanden (*POJO: Plain old Java objects*). Durch die Unterstützung von POJOs in EJB 3.0 ist es möglich, diese mittels Java 5.0 Annotations in Entity Beans zu verwandeln. Diese Entity Beans werden dann vom Application Container (JBoss) verwaltet und über Hibernate serialisiert, in der Datenbank abgelegt und bei Bedarf wiederhergestellt.

Der Zugriff auf diese Entity Beans erfolgt mittels Dependency Injection und der EJB-Query-Language (*ejbQL*), eine an SQL angelehnte Abfragesprache die das Finden und Instanzieren eines Objekts ermöglicht. Hierbei werden die Aufgaben dem Application Container überlassen, was optimieren wie z.B.

Caching verbessern soll.

Schnittstellen

Die Business-Logik (*ControllerBean.java*) besitzt im wesentlichen drei Schnittstellen (siehe auch Grafik „Schnittstellen der Management Konsole“) zur Kommunikation mit den anderen Modulen.

Modul	Zweck	Technik
WSController.java	Kommunikation zwischen dem XFire-Framework und der Session Bean. Ermöglichst den Aufruf von Business-Methoden über SOAP	Suche über JNDI und Aufruf mittels automatisch generierten JNDI-Stubs
JSFController.java	Kommunikation zwischen dem JSF-Framework und der Session Bean. Ermöglichst den Aufruf von Business-Methoden über das Web-GUI	Suche über JNDI und Aufruf mittels automatisch generierten JNDI-Stubs
VMControllerBean.java	Kommunikation mit der VMware Server Virtualisierungsschicht	Da diese Aufrufe lang lauffende Skripte aufrufen , verwenden wir den Java Messaging Services (JMS) zur Entkopplung der Methodenaufrufe

4.1.3 WebGUI (JSF)

Kurzbeschreibung der Komponente

Die WebGUI bietet eine grafische Benutzer-Oberfläche zur Steuerung und Überwachung des Systems. Auf diese GUI kann mit einem beliebigen Web-Browser zugegriffen werden.

Eingesetzte Technologien

Zur Umsetzung dieses Moduls wurden folgende Technologien verwendet:

- Java Server Faces (JSF) – Implementation von Apache myFaces
- Java Server Pages (JSP) als Grundlage für JSF
- Eclipse WTP

Verlauf der Entwicklung

Da wir für die Implementierung der Business-Logik EJB gewählt hatten, bot es sich an, bei der Entwicklung des Web-GUI ebenfalls auf Java-Technologie zu setzen. Um auch hier einen Einblick in die aktuellste Technik zu bekommen, wählten wir die Java Server Faces (JSF).

Eine konkrete Implementation dieses Standards ist myFaces von Apache.org. Die erste Aufgabe war es, das myFaces Framework zu konfigurieren:

- Einbinden der benötigten Bibliotheken als „Web App Libraries"
- Registrierung des Faces Servlets
- Konfiguration der Schnittstelle zwischen JSF-Seiten und der Business-Logik

Das generelle Einbinden der erwähnten Libraries ist einfach, es bedarf allerdings einer Recherche um die Liste der nötigen .jar Dateien zu ermitteln. Der Ansatz alle myFaces Bibliotheken einzubinden funktioniert zwar, sorgt aber für unnötigen Overhead.

Damit das JSF Framework auch für die Verarbeitung von JSP-Seiten verwendet werden kann, muss der Einstiegspunkt - das myFaces Servlet - zunächst im Deployment Descriptor der Web Applikation (*web.xml)* eingetragen werden:

servlet	(((description*, display-name*, icon*)), servlet-name, (servlet-class |
servlet-name	FacesServlet
servlet-class	javax.faces.webapp.FacesServlet
servlet-mapping	(servlet-name, url-pattern)
servlet-name	FacesServlet
url-pattern	/faces/*
servlet-mapping	(servlet-name, url-pattern)
servlet-name	FacesServlet
url-pattern	*.jsf

Das Konzept von JSF macht es dem Programmierer sehr leicht, schnell eine einfache Seite zu entwickeln. Die Schnitstelle zwischen der JSF-Seite und der restlichen Java-Welt heißt im JSF Kontext *Managed Bean*. Diese Managed Bean wird innerhalb des Desployment-Descriptors *faces-config.xml* deklariert. Danach kann innerhalb der JSF-Seite auf diese Java-Klasse über den angegebenen Alias zugegriffen werden:

faces-config	((application | factory | component | conv
navigation-rule	(description*, display-name*, icon*, from
managed-bean	(description*, display-name*, icon*, mana
managed-bean-name	controller
managed-bean-class	de.tmxm.wef.jsf.JSFController
managed-bean-scope	session

Wurde als Scope „*Session*" gewählt bekommt jeder Benutzer der Seite eine eigene Instanz dieser Managed Bean. Hier kann somit der State des Benutzers gehalten werden. Auch die Daten des aktuellen Views werden hier hinterlegt.

Um die für unseren View notwendigen Daten zu erhalten wird innerhalb der Controller-Klasse (*JSFController.java*) mittels JNDI eine Stub-Klasse auf die Business-Logik (*ControllerBean.java*) aufgerufen. Hierdurch erhalten wir zusätzlichen Zugriff auf alle Business-Methoden um z.B. neue URLs oder VMs anzulegen bzw. zu löschen.

Schnittstellen

Die mittels JSF-Tags erstellten Seiten erhalten eine Referenz auf die in der *faces-config.xml*eingetragenen Beans (JSFController.java) und dadurch auch Zugriff auf die Business-Logik. Siehe auch *Schnittstellen der Business-Logik.*

Mögliche Erweiterungen

Angedachte Erweiterungen für die nächste Version:

- AJAX – Zur automatischen Aktualisierung des Views und zur vereinfachten Dateneingabe
- Weitere Optionen zur Konfiguration des Systems
- Import-Funktion für große URL-Listen

- Möglichkeit zur Abfrage von Screenshots der aktiven virtuellen Maschinen (siehe auch *Erweiterungen der VM Komponente*)
- Ansprechenderes Design und Layout

4.1.4 SOAP Web-Services (XFire)

Kurzbeschreibung der Komponente

Da es sich bei unserem System um eine verteilte Architektur handelt, benötigen wir an einigen Stellen eine Kommunikationsschnittstelle. Für die Kommunikation zwischen dem BrowserControl und der Management-Konsole setzen wir auf die WebServices-Technologie auf Basis von SOAP (*Simple Object Access Protocol)*.

Eingesetzte Technologien

Zur Umsetzung dieses Moduls wurden folgende Technologien verwendet:

- XFire Framework – SOAP Webservices Framework
- WSDL – Web Service Description Language
- Eclipse WTP

Verlauf der Entwicklung

Bereits zu Beginn des Projekts begannen wir mit der Recherche nach einem geeigneten Kommunikationsprotokoll bzw. Serialisierungsverfahren.

Zur Auswahl standen:

- Java RPC
- CORBA
- WebServices SOAP

Da es unwahrscheinlich war, dass auch der Client in Java programmiert werden würde, schied die reine Java Lösung sehr früh aus. Nach der Evaluierung von CORBA stellte sich auch diese Lösung als nicht brauchbar heraus, da die alleinige Einarbeitung in die Technik den zeitlichen Rahmen der Komponente und des Projektes sprengen würde.

Die Wahl fiel deswegenauf WebServices auf Basis von SOAP. Die nächste Hürde war es eine der zahlreichen SOAP-Frameworks auszuwählen. Der erste Favorit war das JBoss-eigene Framework, welches aber noch auf EJB in der Version 2 setzt. Da uns auch diese Lösung als zu umständlich erschien, da sie nicht ohne Weiteres mit unserer in EJB 3.0 realisierten Business-Logik kompatibel war, musste eine andere Lösung gefunden werden. Der naheliegendste Ansatz - einfach die EJB 3.0 kompatible JBoss-Lösung zu verwenden - scheiterte leider an der frühen Betaphase des Frameworks.

Nach weiteren Recherchen stießen wir auf das XFire Framework, das

WebServices auf Basis von Java-POJOs realisiert. Da dies perfekt in das Konzept unserer Business-Logik passte, wählten wir nach einigen Tests dieses Framework aus.

Der erste Entwicklungsschritt war identisch zum JSF-Framework. Das „Eingangs"-Servlet musste zunächst der Web-Applikation bekannt gemacht werden. Dies geschieht in der *web.xml* Deployment Descriptor Datei. Zusätzlich werden noch eine oder mehrere Servlet-Mappings erstellt, die den Pfad welcher eine Ausführung des Servlets triggert, konfiguriert:

servlet	(((description*, display-name*, icon*)), servlet-name, (servlet-
servlet-name	XFireServlet
servlet-class	org.codehaus.xfire.transport.http.XFireConfigurableServlet
servlet-mapping	(servlet-name, url-pattern)
servlet-name	XFireServlet
url-pattern	/servlet/XFireServlet/*
servlet-mapping	(servlet-name, url-pattern)
servlet-name	XFireServlet
url-pattern	/services/*

Um nun bereits vorhandene Java-Methoden als WebServices anbieten zu können genügt es, ein weiteres Java-Interface zu erzeugen welches alle relevanten Methoden umfasst. Dieses Interfaces wird dann als *serviceClass* in den Deployment-Descriptor des Frameworks eingetragen (*services.xml*). Auch die Klasse, die dieses Interface implementiert, wird angeben (*implementationClass*). Danach erhält der WebService noch einen Namen und einen Namespace um einen eindeutigen Bezeichner zu erhalten:

beans	
xmlns	http://xfire.codehaus.org/config/1.0
service	
name	wef
namespace	tmxm
serviceClass	de.tmxm.wef.ws.IWSController
implementationClass	de.tmxm.wef.ws.WSController

Wird jetzt von einem SOAP-Client die URL

http://localhost:8080/services/wef

aufgerufen erzeugt das Framework die nötigen Proxy-Klassen, um die SOAP-Aufrufe auf das angegebene Interface zu mappen. Es ist also denkbar einfach, weitere Methoden als WebService anzubieten. Für die Erstellung des SOAP-Clients werden in der Regel auch Frameworks verwendet, welche die nötigen Stubs/Proxies aus der WSDL-Datei generieren. Um diese Datei zu erhalten genügt folgender Aufruf:

http://localhost:8080/services/wef?wsdl

Schnittstellen

Um mit der Business-Logik kommunizieren zu können wurde die Java-Klasse *WSController.java* erstellt. Dieses Klasse enthält alle Methoden der Business-

Logik, die als WebServices angeboten werden sollen. Der Aufruf erfolgt über JNDI-Stubs analog zur den JSF-Schnittstellen.

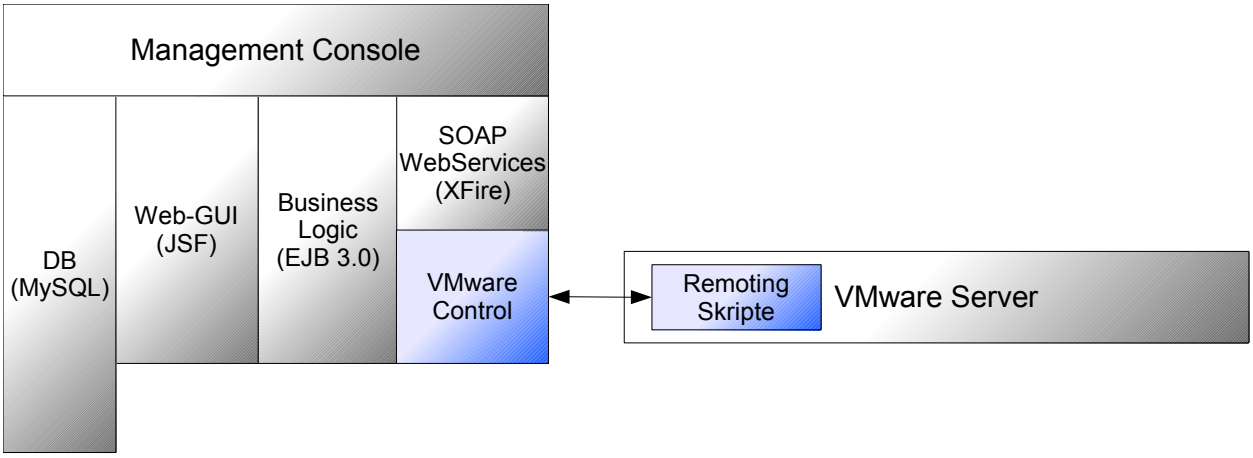
Mögliche Erweiterungen

Ein kleines Manko der jetzigen Lösung ist die Tatsache, dass Aufrufe nur in eine Richtung möglich sind, und zwar nur von der BrowserControl-Komponente zur Management-Konsole. Für die jetzigen Aktionen (Registieren, neue URLs abfragen, Konfiguration abfragen) reicht diese Funktionalität.

Da es später aber auch möglich sein sollte, in einer virtuellen Maschine auf „Zuruf" eine Aktion auszuführen sollte auch der „Client" WebServices anbieten können.

4.1.5 VmControl Java Modul und VMware Skripte

Übersicht



Kurzbeschreibung der Komponente

Die VMware-Skripte stellen die Verbindung zwischen der Management-Konsole und dem VMware Server (bei uns *VM-Wirt* benannt), also der Host-Maschine auf der alle Virtuelle Machinen (VMs) registriert sind und im Betrieb sind, dar. Auf der Management-Konsole steht dann eine Java-Klasse "VMControlerBean.java" bereit, die die Brücke zur Business-Logik im Management-Flow ist.

Eingesetzte Technologien

Nach kurzer Evaluierung war klar, dass wir auf die kurz zuvor kostenlos freigegebene VMware-Software *VMware Server* bauen konnten.

Siehe *http://www.heise.de/newsticker/meldung/69235*

Diese besitzt neben dem Server-Modul ein bereits vorkompiliertes RPM-Paket für unsere Linux-Distribution, ebenso einen Windows- und Linux-Client, um die einzelnen virtuellen Maschinen von einer entfernten Maschine aus bedienen zu können.

Aufgrund der vorhandenen Linux-Erfahrung und einem sehr guten, bereits vorhandenen Kommandozeilen-Programm (*vmware-cmd*) für die Steuerung der VMware – Software war sehr schnell klar, dass der Großteil unserer Programme auf dem VM-Wirt in der Skript-Sprache *bash* geschrieben wird. Jedoch besitzt *vmware-cmd* nicht alle Funktionen, die beispielsweise die Remote-GUI-Client-Komponente von VMware-Server zur Verfügung stellt.

Es ging uns vor allem um die Erstellung eines Snapshots (einer Momentaufnahme einer virtuellen Maschine, zu der zu späterem Zeitpunkt zurückgekehrt werden kann) sowie die Rückkehr zu diesem erstellten

Snapshot ("revert to snapshot"). Diese Fähigkeiten sind lediglich über die von VMware zur Verfügung gestellte C-Schnittstelle ***VIX*** verfügbar.

Einarbeitung *http://www.vmware.com/pdf/Server_C_API_Programming.pdf*

Deshalb musste ein Teil unserer VMware-Komponente in C programmiert werden.

Ausserdem benötigten wir eine Möglichkeit, dass unsere Mangement-Konsole die Skripte anstößt und auch eine Antwort (beispielsweise die IP-Adresse einer neu erstellen virtuellen Maschine) erhalten kann, ohne dabei den gesamten Management-Flow zu blockieren. Nach reiflicher Überlegung entschlossen wir uns für eine simple Variante, nämlich die Ausführung eines Programmes über *SSH*, welches von der besagten Java-Klasse (*VMControl.java*) ausführt. Die einzelnen Komponenten dieses Bausteines sind wie folgt definiert:

VMControl.java **VMControl Klasse**	Teil des Business – Logik auf der Management-Konsole Stößt Shell-Skripte per SSH an Benutzt Java's Runtime & Process-Interfaces
VM-Skripte	Fünf Bash-Skripte, die die einzelnen, benötigten Funktionen von vmware-cmd und VMControl (das C-Programm) zusammenfügen, MAC-Adressen berechnen und die VM-Dateien vom Prototyp kopieren.
VMControl	C-Programm, wird benötigt um die Funktionalität von vmware-cmd um Snapshot-Fähigkeit zu erweitern und das Kopieren sowie das Starten von BrowserControl im Gast-System möglich zu machen.
vmware-cmd	Das bereits vorhandene Kommandozeilen-Programm von VMware für das Starten einer virtuellen Maschine mit beschränkter Funktionalität

Benötigte Features der VM-Skripte

Zuerst einmal mussten wir uns überlegen, welche Steuerelemente wir vom VM-Wirt benötigen. Wir kamen auf fünf Hauptaufgaben für die Skripte:

1. Neu-Aufsetzen einer virtuellen Maschine (*vmclone*)
2. Zurückkehren zu einem "sauberen" Zustand einer virtuellen Maschine

(*vmrevert*)

3. Löschen einer virtuellen Maschine (*vmremove*)
4. Initialisierung unserer BrowserControl-Applikation auf einer virtuellen Maschine (*vminstall*)
5. Auflistung aller bereits vorhandenen virtuellen Maschinen (nicht den Prototyp) (*vmlist*)

Entwicklungsschritte

vmcontrol (C-Programm)

Das C-Programm ergänzt die Funktionalität, die bei vmware-cmd fehlt und nur durch die C-API von VMWare (VIX) möglich ist. Diese sind im Wesentlichen

- das Erstellen von Snapshots - *Vix_CreateSnapshot()*
- das Zurückkehren eines bestehenden Snapshotes *Vix_RevertToSnapshot()*
- das Kopieren einer Datei vom VM-Wirt zu einer virtuellen Maschine *Vix_CopyFileFromHostToGuest()*
- sowie das Ausführen eines Programmes auf einer virtuellen Maschine *Vix_RunProgramInGuest()*

DasC-Programm *vmcontrol* wird als Subkomponente unserer Bash-Skripte benutzt. Wir entschieden uns dazu, weil die bash-Programme leichter zu warten, schneller zu debuggen und leicht erweiterbar sind. Anfänglich hatten wir Schwierigkeiten beim Compilieren unseres C-Programmes, was sich aber nur als Fehl-Konfiguration der Make-Files herausstellte.

vmclone.sh

Wir entschieden uns für eine *Prototyp-Cloning* - Methode (siehe 3.2.1), bei der wir eine saubere Prototyp-VM kopieren, die MAC-Adresse für die neue virtuelle Maschine setzen und die VM danach starten wollten. Sobald die virtuelle Maschine gestartet wurde, soll das C-Programm *vmcontrol* einen Snapshot des Rechners erzeugen, damit wir jederzeit dorthin zurückkehren können, falls eine schadhafte Webseite den Rechner komprimittiert hat.

VMware identifiziert die einzelnen virtuellen Maschinen anhand einer Konfigurationsdatei (z.B. *winxp_client001.vmx*) im Datenverzeichnis der jeweiligen virtuellen Maschine. In ihr stehen die virtuellen Komponenten der Maschine (ob z.B. nun ein CD-Laufwerk für die VM angeschlossen ist oder die Netzwerkkarte aktiv ist oder das „virtuelle Kabel" ausgesteckt wurde). Diese sollte man mit vmware-cmd ändern können (beispielsweise die MAC-Adresse der virtuellen NIC), doch mussten wir die Datei dann letztendlich mit bash selbst bearbeiten, da vmware-cmd dort offenbar einen Bug besitzt.

Die einzelnen Schritte unseres vmclone-Programmes sehen wie folgt aus:

1. Laden der zuletzt erstellten VM, um die MAC-Adresse auszulesen und eine neue zu generieren
2. Die Daten der Prototyp-VM kopieren (Festplatten-Datei & Konfiguration)
3. Konfiguration ändern (MAC-Adresse eintragen; Neue VM mit vmware-cmd beim VMware Server registrieren; Neuer Name eintragen)
4. VM starten und warten bis Windows XP gebootet ist. Danach mit dem C-Programm den Benutzer bei Windows XP anmelden und den Snapshot generieren
5. Ausführen des *vminstall* – Skriptes
6. Ausgabe der IP-Adresse (benötigt für unsere Management-Applikation)

Besonders das Kopieren der Prototyp-Daten stellte sich als sehr zeitintensiv dar, da schließlich 2 Gigabyte Daten beim Cloning-Prozess kopiert werden müssen. Leider ist dies fast unvermeidlich,wir haben aber bereits Workarounds konzipiert, die aus Zeitmangel nur als mögliche Erweiterung unter "**Mögliche Erweiterungen**" am Ende dieses Kapitels aufgeführt sind.

vmrevert.sh

Das Skript ist im Gegensatz zum vmclone-Skript (welches das Mächtigste der bash Skripte ist) einfacher und schneller ausgeführt. Es erhält als Parameter die IP-Adresse der virtuellen Maschine und sucht sich die entsprechende Konfigurationsdatei aus. Danach wird das C-Programm *vmcontrol* mit dem Parameter "revertToSnapshot" gestartet.

Zum Abschluss wird wie bei *vmclone.sh* das Skript *vminstall.sh* gestartet, um den Flow-Prozess des BrowserControls zu starten.

vmremove.sh

Es werden die Konfigurationsdateien mit einfachen Linux-Befehlen gelöscht (*rm -rf*) sowie die virtuelle Maschine am Server "unregistriert". Als Parameter wird die entsprechende IP-Adresse benötigt.

Ein Beispiel zur Ausführung wäre also (analog bei den anderen Skripten), welches von der Java-Klasse über SSH ausgeführt wird:

/pfad/zu_den/skripten/vmremove.sh 192.168.1.51

vminstall.sh

Hierbei wird die aktuelle Version des BrowserControls (*BrowserControl.exe*) vom VM-Wirt auf das Gast-System kopiert. Danach wird diese Datei auf dem Gast-System ausgeführt.

Dies ist lediglich ein Wrapper auf die vmcontrol-Funktionen, die wir mit der C-API geschaffen haben. Dennoch bereiteten uns diese Methoden der API

größere Probleme, da wir auf unseren virtuellen Maschinen keine Passwörter für die Benutzer hatten und die Methoden zwingend ein Login vorschreiben, um Programme zu kopieren oder ausführen.

Nach etwas Recherche im Internet konnten wir dann ein Auto-Login in Windows XP durchführen sowie ein Passwort setzen und die Programme dann doch ausführen. Das *vminstall.sh* - Skript in seiner jetzigen Form bietet somit auch ein gelungenes Deploy-System, damit die virtuellen Maschinen jederzeit eine aktuelle Version von *BrowserControl.exe* erhalten (falls es dort Updates gibt) und man nicht die Dateien manuell auf jede virtuelle Maschine kopieren muss.

vmlist.sh

Um einen Überblick über die aktuellen Instanzen zu erhalten, gibt dieses Skript alle geklonten virtuellen Maschinen aus. Dazu wird eine vmware-cmd Funktion benutzt, die allerdings alle virtuellen Maschinen (auch den Prototypen oder Testsysteme) ausgibt. Mit einigen Filtern erhielten wir dann nur die IP-Adressen der geklonten Instanzen. vmlist wird auch intern für *vmclone.sh* benutzt, um die nächsthöhere MAC-Adresse für eine neue VM zu erhalten.

Ziel unserer Skripte war auch die Transparentheit der Adressen-Schichten. Dies bedeutet vor allem, dass der Benutzer der Skripte lediglich die IP-Adressen einer virtuellen Maschine benötigt, obwohl VMware intern nur mit "Konfigurationsdateien" arbeitet und MAC-Adressen benötigt. Jedoch bleibt dies für den Benutzer der Management-Konsole unbemerkt, was mehr Klarheit bei der Bedienung schafft.

VMControl.java

Die Java-Klasse wird auf der Management-Konsole benötigt. Sie stellt in der Business-Logik die Schnittstelle zwischen den Shell-Aufrufen und den VMObject-Instanzen dar, die dann in der Applikation abgelegt werden. Da Java keine ganz einfache Möglichkeit bietet, Programme auszuführen, mussten wir uns auch wieder dem Web bedienen um diverse Tutorials zu finden. Sie spiegelt auch die verwendeten Funktionen wider, da sie für jedes Skript eine Methode besitzt (*vmclone.sh* => VMControl.vmClone() gibt ein neues VMObject zurück und enthält dann die neue IP-Adresse, die das Skript zurückgibt). Letztendlich führt die VMControl-Klasse einen transparenten SSH-Aufruf aus (*ssh 192.168.1.2 /data/vmcontrol/vmclone.sh*)

SSH-Brücke

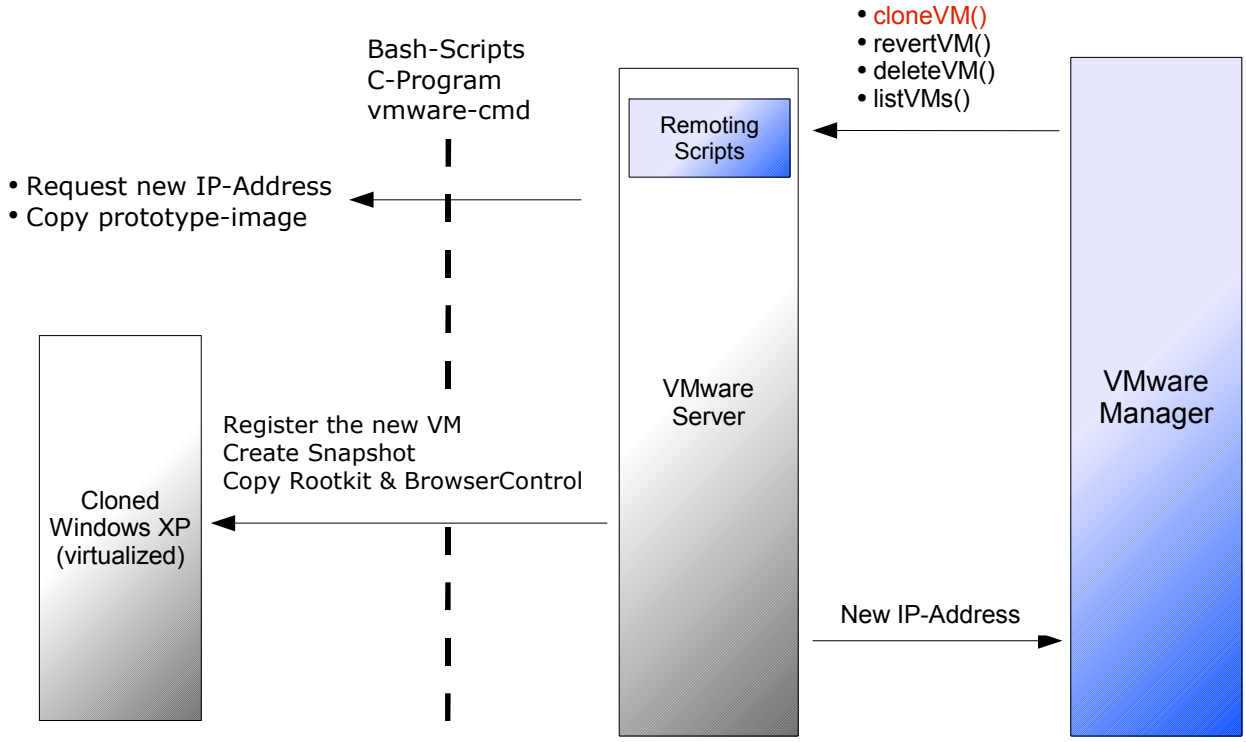
Anstatt ein eigenes Protokoll zu erfinden, benutzten wir das sichere Protokoll SSH für unsere Remote-Aufrufe. Theoretisch wäre es auch möglich gewesen, unsere Skripte von der Management-Konsole aus zu starten und uns dann per remote vmware-cmd unsere einzelnen Befehle auszuführen. Jedoch ist

dieser vmware-cmd Schritt von VMware sehr beschränkt, es ist dann z.B. nicht möglich, Dateien vom Host auf das Gast-System zu kopieren (siehe *vminstall.sh*).

Dennoch hat uns das Ergebnis dieser Idee, lediglich SSHzu benutzen, sehr positiv überrascht, da sich die Skripte nahtlos in die Business-Logik integrierten und alles transparent funktioniert.

Um bei SSH auf das Public-Key-Verfahren zurückzugreifen, mussten wir auf der Management-Konsole ein Schlüsselpaar ohne Passphrase erzeugen (*ssh-keygen -t dsa -b 1024*), und den Public Key als "authorized_keys"-Datei auf den VM-Wirt in das entsprechende SSH-Verzeichnis legen. Nun konnten wir SSH als transparente Brücke schlicht und schnell nutzen.

Schnittstellen



Mögliche Erweiterungen

Während der Entwicklung stießen wir an so manche Grenzen, vor allem was den Zeitaufwand betraf. Sehr gerne hätten wir eine Screenshot-Funktion eingebaut, um im Web-Interface einen aktuellen Bildschirm der virtuellen Maschine sehen zu können.

Die Deployment-Funktion (siehe *vminstall.sh*) durch das Kopieren der BrowserControl.exe – Datei auf das Gast-System nach jedem Zurücksetzen einer virtuellen Maschine konnten wir letztlich noch einbauen, da es lediglich ein Mehraufwand von zwei Stunden war und uns diverse andere Probleme ersparte.

Momentan dauert das *vmclone.sh*-Skript noch über 2 Minuten, weil eben 2

GB Festplatten-Daten der virtuellen Maschine kopiert werden müssen. Hier wäre auch ein besseres Konzept wünschenswert.

Ebenso gilt die Einschränkung, dass das Klonen momentan nur sequenzell funktioniert, also nicht mehrere virtuelle Instanzen gleichzeitig aufgesetzt werden können.

4.2 VM Wirt

4.2.1 Server-Installation

Wir benutzten für die Installation des Servers selbst (ebenso für die Management-Konsole) die frei erhältliche Linux-Distribution „CentOS" [http://www.centos.org], die ein community-basierter Ableger von RedHat Enterprise Linux (RHEL) ist. Die Installation verlief ohne Probleme, da wir in diesem Gebiet sehr viel Erfahrung hatten. Des weiteren konnten wir das VMware – Paket als RPM (Redhat Package Manager) ohne Probleme installieren und benutzen. Auf einer separaten Partition für unsere virtuellen Maschinen konnten wir dann sowohl den Windows – Prototyp installieren als auch Testrechner für die Entwicklung des Rootkits aufsetzen. Die Steuerung funktioniert über ein Client, welches von VMware für Windows und Linux zur Verfügung gestellt wird.

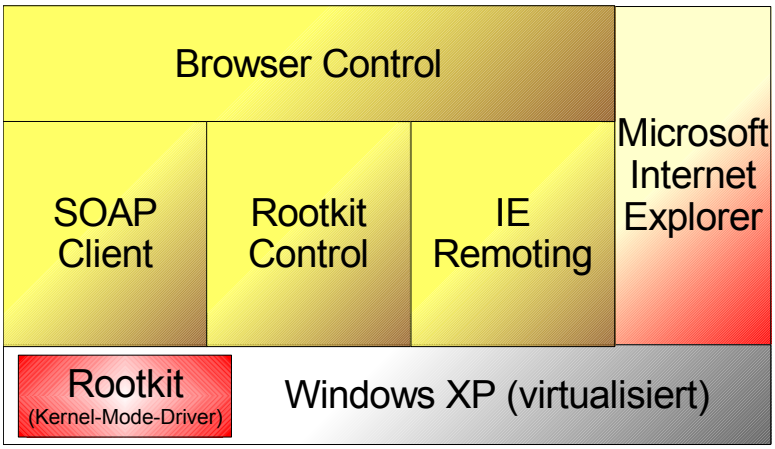
4.2.2 Windows XP Image Template (Prototyp)

Für die Virtualisierung war von Beginn an klar, dass wir das Prinzip des *Klonens* einsetzen, also eine installierte Windows-Version für jede Instanz kopieren. Die ursprüngliche Windows-Installation sollte ein ungepatchtes Windows XP Professional sein. Ohne Service Pack 2 oder Updates fällt es uns somit leichter gefährliche Webseiten zu entdecken.

Weiterhin setzten wir das Programm *XPLite* ein, das uns erlaubte, die bestehenden Windows-Programme und die während einer üblichen Windows-Laufzeit geladenen Systemdateien auszuschalten und somit die Windows – Installation schneller und schlanker zu machen. Dies verringerte das Performance – Problem, sowohl beim Klonen und beim Starten einer Windows-Instanz als auch beim gleichzeitigen Betrieb von mehreren virtuellen Maschinen, da diese dann weniger Arbeitsspeicher zur Laufzeit benötigen. Somit können mehr virtuelle Instanzen gleichzeitig laufen.

4.3 Browser Control

Übersicht



Kurzbeschreibung

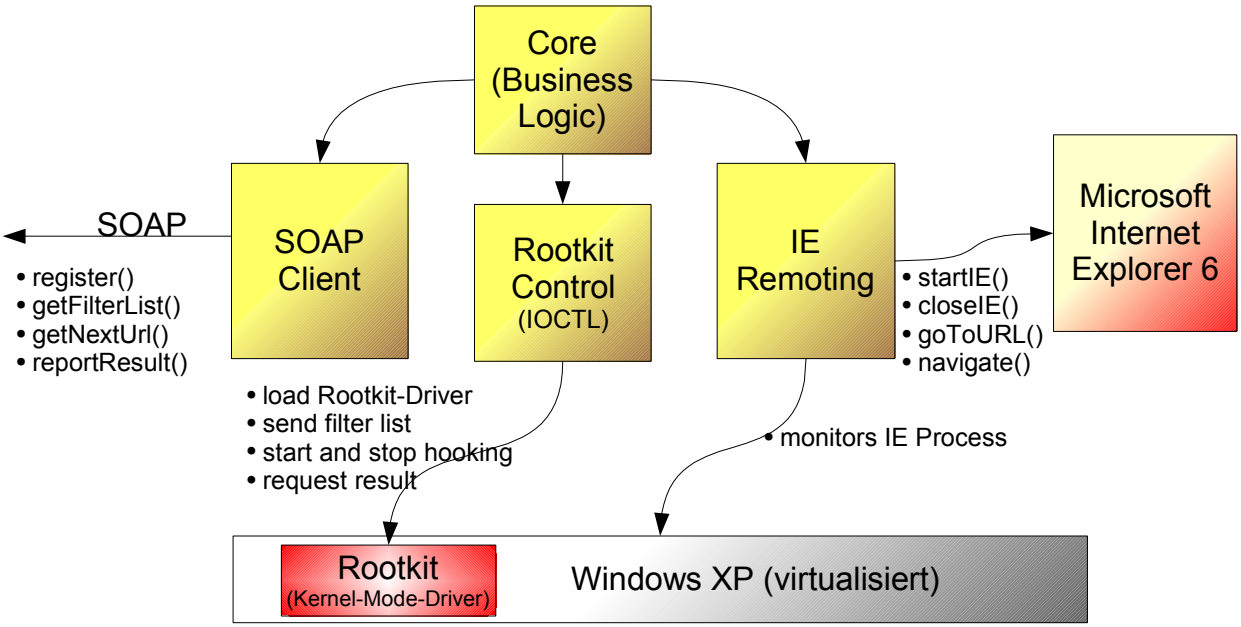
Das Browser-Control bildet das Gegenstück zur Management-Konsole. Es ist pro virtualisierte Windows XP - Instanz einmal vorhanden und erfüllt folgende Aufgaben:

- Kommunikation mit der Management-Konsole
- Steuerung des Windows Kernel Rootkits
- Steuerung des Microsoft Internet Explorer

Für jede dieser Aufgaben wurde ein eigenes Modul erstellt, um die Entwicklungsschritte parallel durchführen zu können, was aber im Laufe des Projekts zu einigen Problemen führte. Details hierzu bei den Entwicklungsschritte der einzelnen Module.

4.3.1 Browser Control Schnittstellen

Übersicht



4.3.2 Rootkit-Control

Kurzbeschreibung der Komponente

Das *Rootkit-Control* ist die zentrale Komponente des Browser-Controls. Es verbindet den SOAP-Client und das IE-Remoting-Modul, die beide als C++ Library entwickelt wurden. Im Bezug auf das Rootkit im Windows-Kernel hat es folgende Aufgaben:

- Laden des Rootkits als Windows System-Treiber
- Deaktivieren des Rootkits
- Konfiguration des Rootkits – auf Basis der mittels SOAP angeforderter Konfiguration
- Aktivieren des Hookings (Umleiten von Systemcalls)
- Deaktivieren des Hookings
- Abfragen der Ergebnisse

Eingesetzte Technologien

Da das Rootkit in der Programmiersprache C entwickelt wurde, kommen für die Entwicklung des Rootkit-Control folgende Technologien zum Einsatz:

- C++ Windows Programmierung

- IOCTL Programmierschnittstelle zur Userspace-Kernel - Kommunikation
- Microsoft Visual Studio 2005

Verlauf der Entwicklung

Nachdem eine erste Version des Kernel-Rootkits zu Beginn dieser Entwicklung bereits vorhanden war, stellte sich die Frage in welcher Programmiersprache diese Steuerung entwickelt werden sollte.

Zur Auswahl standen hierbei:

- Microsoft C++
- Visual Basic
- C# und .NET

Die .NET - Lösung schied bereits sehr früh aus, da wir den Overhead der .NET Runtime vermeiden wollten um unser Windows XP Image möglichst klein zu halten.

Die Entscheidung für C++ und gegen Visual Basic war nicht technisch begründet, da beide Programmiersprachen mittels der IOCTL-Schnittstelle kommunizieren können. Es war viel mehr eine Entscheidung aufgrund unserer Vorkenntnisse.

Das „Problem" bei der Kommunikation mit Treibern im Kernel war, dass diese nicht selbständig eine Userspace-Applikation aufrufen konnten. Deshalb muss jede Kommunikation vom Rootkit-Control eingeleitet werden.

Nachdem die Entwicklungsumgebung in Visual Studio eingerichtet war, begann die Entwicklung anhand von Beispielen aus dem Internet. Zunächst wurden Methoden entwickelt, die es ermöglichen während der Laufzeit des Betriebssystems einen neuen Gerätetreiber zu laden. Danach musste eine Möglichkeit zur Kommunikation gefunden werden. Hierbei gab es zum einen die Möglichkeit den Treiber direkt über *read()*- und *write()*-Systemcalls anzusprechen (als wäre es ein Filesystem-Treiber). Dies hat aber den Nachteil, dass wir auf beiden Seiten nur Datenblöcke ohne Semantik erhalten. Da wir nicht die Zeit hatten uns unserer eigenes Binär-Protokoll zu entwerfen, entschieden wir uns für die Verwendung der *IOCTL-Funktion*.

Dadurch war es möglich, für jede gewünschte Aktion (z.B. das Starten des Hookings) eine eindeutige ID zu erzeugen, die dann im Kernel entsprechend behandelt werden konnte.

Der Aufruf von Aktionen, die keine Rückgabe aus dem Kernel bzw. das Senden von Daten in den Kernel erwarten, funktionierten sehr schnell. Alle Aufrufe wie das Senden der Konfiguration oder das Abfragen der Ergebnisse waren deutlich aufwändiger und kosteten uns viel Zeit.

Der Transport von Daten in den oder aus dem Kernel ist in Windows über Shared Memory realisiert, d.h. während des entsprechenden Systemcalls kann das Rootkit-Control Daten in diesen Shared Memory schreiben und lesen. Da diese Datenstrukturen aber nur für einen sehr kleinen Zeitraum gültig bleiben, müssen sie sowohl im Kernel als auch im Userspace kopiert werden. Genau diese Punkte im Programm zu finden und zu ändern kostete die meiste Zeit während der Entwicklung und es bestand auch immer die Gefahr, durch unnötiges Hin- und Herkopieren Memory-Leaks zu erzeugen.

Ein weiteres Problem war die Art und Weise wie Daten gelesen oder geschrieben werden können. Es ist nur möglich einen zusammenhängenden Datenblock mit bekannter Länge zu verwenden. D.h. es ist beispielweise nicht möglich, die Konfiguration als String-Array in den Kernel zu schicken, da zum einen die einzelnen Strings nicht zwangsläufig nacheinander im Speicher liegen und es auch nicht ohne weiteres möglich ist, die Größe in Bytes zu ermitteln. Unsere Pläne für die Kommunikation eigene Datenstrukturen wie z.B. verkettete Listen zu verwenden scheiterten.

Es musste also jede Konfigurationszeile einzeln in den Kernel geladen werden. Beim Abrufen der Ergebnisse musste zusätzlich noch vor dem eigentlichen Kopieren der Daten die Länge der Ergebnisliste abgefragt werden.

Schnittstellen

Die Schnittstelle in den Kernel bildet das IOCTL-Interface. Für die Kommunikation mit dem SOAP-Client und dem IE-Remoting wurden diese beiden Module als Library exportiert und im Rootkit-Control als C++ Object wieder eingebunden. Dadurch kann jedes Modul separat weiter entwickelt werden. Bei einer Änderung an einer Komponente muss jedoch die gesamte Applikation neu übersetzt werden.

Mögliche Erweiterungen

Da die Konfiguration des Rootkits in einer der nächsten Versionen deutlich komplexer werden wird, ist eine Erweiterung und Überarbeitung des jetzigen Quelltextes nötig. Zusätzlich wäre es denkbar, die Applikation von einer Konsolen-Applikation in einen Windows Service umzuwandeln.

4.3.3 Rootkit

Kurzbeschreibung der Komponente

Unser Rootkit ist eine Modifikation des Windows-Kernels um verdächtige Systemaktivitäten erkennen zu können. Die Idee hierbei ist folgende: Wenn die schadhafte Seite dauerhaft einen Rechner übernehmen möchte, muss sie ihn in irgendeiner Form modifizieren. Denkbar wäre das Erstellen neuer Dateien und das Anlegen von neuen Keys in der Windows Registry. Auch das plötzliche Starten einer Applikation gilt als verdächtig. Deshalb überwacht und protokolliert unser Rootkit die korrelierenden Systemaufrufe und löst bei Bedarf Alarm aus.

Eingesetzte Technologien

Bei der Entwicklung kamen folgende Technologien zum Einsatz:

- Die Programmiersprache C
- Das Windows Driver Development Kit (DDK)
- Externe Buildskripte
- VMware Server

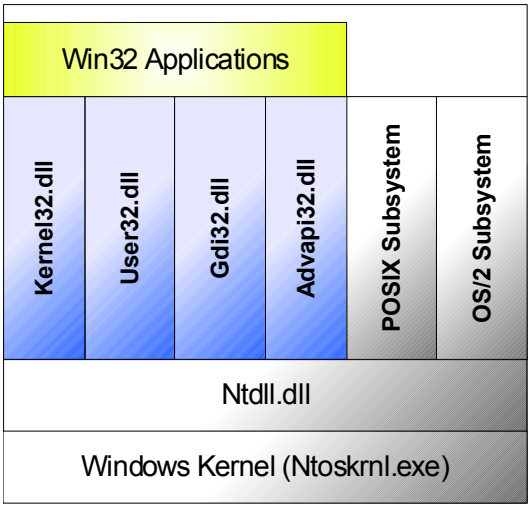
Verlauf der Entwicklung

Die Entwicklung des Rootkits war für alle Teammitglieder Neuland. Zwar waren bereits Erfahrungen mit der Programmiersprache C vorhanden, wie sich aber im Laufe des Projektes feststellte hat die Entwicklung eines Rootkits nur sehr wenig mit der einer normalen C-Anwendung gemein.

Zunächst arbeiteten wir uns mittels umfangreicher Internet Rechere und zahlreichen Whitepapern in die Thematik der Rootkits ein.ein. Nach einigen Tagen hatten wir das theoretische Wissen über die unterschiedlichen Möglichkeiten Systemcalls umzuleiten. Aus der Menge der Möglichkeiten kamen zwei in die engere Auswahl:

- Import Address Table Hooking (IAT)
- System Service Table Hooking (SST)

Um diese beiden Verfahren zu verstehen und eine Auswahl zu treffen, mussten wir uns zunächst mit dem Aufbau von Windows aus Sicht der Applikation beschäftigen.

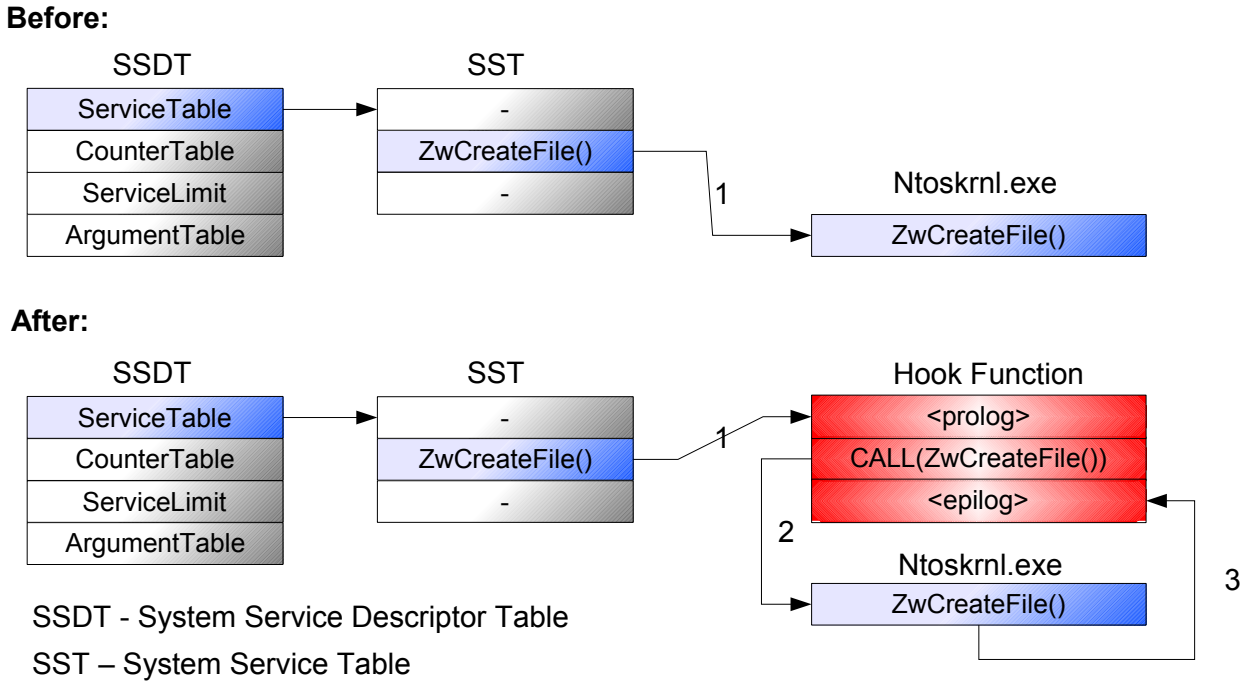


Um die Systemunabhängigkeit und die Abwärtskompatibilität von Windows-Applikationen zu gewährleisten, setzt Microsoft auf mehrere Abstraktionsschichten. Ganz unten ist der Betriebssystem-Kern, der je nach Windows-Version unterschiedlich ist. Er enthält die eigentliche Implementierung der Systemcalls,im Falle von Windows XP heißt dies z.B. *ZwCreateFile()*. Die nächste Schicht bildet die Datei *Ntdll.dll* - sie enthält die selben Systemcalls in der Form *NtCreateFile()* und leitet diese nach eventuellen Anpassungen an den Kernel weiter.

Auf dieser Schicht setzten dann die einzelnen Subsysteme auf. Das Bekannteste davon dürfte das Win32-Subsystem sein. Es besteht aus mehreren System-DLLs, die dem Applikationsentwickler eine einfache und vor allem auf allen Windows Versionen identische Programmierschnittstelle bieten. Ähnlich der *Ntdll.dll* leiten auch diese Schichten ihre Aufrufe weiter und führen nur einige Anpassungen durch. In dieser Schicht haben die Systemcalls die einfache Form *CreateFile()*.

Möchte man also einen Systemcall umleiten, gibt es die Möglichkeit eine der Win32-DLLs gegen seine eigene DLL auszutauschen (IAT-Hooking), welche die Aufrufe überwacht und dann an die eigentliche DLL weiterleitet. Dies geschieht, indem dem Programm beim Start eine Referenz auf die veränderte DLL übergeben wird. Der entscheidende Nachteil hierbei ist jedoch, dass dies für jede Applikation einzeln durchgeführt werden muss und dass man auch nur Aufrufe der Win32-API überwachen kann. Verwendet ein Angreifer die versionsspezifischen Systemcalls - z.B. *NtCreateFile()* - würde dies nicht erkannt.

Aus diesem Grund haben wir uns für das SST-Hooking entschieden. Bei der Weiterleitung von Aufrufen zwischen der *Ntdll.dll* und dem Windows-Kernel verlässt sich das Betriebssystem auf eine Datenstruktur namens ***SST*** (*System Service Table*). In ihr steht die Speicheradresse für jeden Systemcall an der dieser Systemcall aus der *Ntoskrnl.exe* geladen wurde.



Das Ziel ist es also, die Speicheradresse mit einer anderen Adresse zu überschreiben. Dies ermöglicht es dem Rootkit, den Systemcall auf die eigene Implementation umzulenken und zu überwachen bzw. zu verhindern. In der Regel ruft das Rootkit selber den gewünschten Systemcall auf, um die Funktionsweise des Systems nicht zu verändern. Diese Methodik ist sehr mächtig, da es nun kaum noch eine Möglichkeit gibt, der Überwachung zu entgehen.

Da sich die erwähnte Datenstruktur im Speicherbereich des Windows-Kernel befindet, ist es nicht möglich, diese Modifikation mit einer normalen Windows-Applikation durchzuführen. Eine Möglichkeit, den Zugriff auf Kernelspeicher zu erlangen, ist die Entwicklung eines eigenen Gerätetreibers.

Die erste Aufgabe war es also, uns mit der Windows-Treiber-Entwicklung zu beschäftigen. Zunächst richteten wir eine Entwicklungsumgebung ein. Zur Entwicklung von Systemtreibern bietet Microsoft ein spezielles Werkzeug, das Windows Driver Development Kit (DDK) an. Dieses Kit enthält die notwendigen Libraries, Header-Dateien und einen speziellen Compiler. Jedoch ist eine Integration in Visual Studio .NET nicht vorgesehen, was die Entwicklung deutlich erschwert, da ohne eine IDE programmiert und mittels Kommandozeile übersetzt werden muss.

Nach einigen Tagen Entwicklung fanden wir dann eine Sammlung von Skripten, die eine Integration in Visual Studio ermöglichten und den Übersetzungsvorgang automatisierten.

Mit der Hilfe dieser Skripte und einigen Beispielen aus dem Internet konnten wir schnell die ersten Erfolge verzeichnen. Die Entwicklung der treiber-spezifischen Teile verlief daher recht zügig. Auch das eigentliche Hooking,

also die Manipulation der SST, war nicht sonderlich zeitaufwendig, da wir den nötigen Quelltext aus einem frei verfügbaren Rootkit entnahmen und anpassten. Auf Grund des engen Zeitplans setzten wir im Rahmen des Software Projekts wir nur die *NtCreateFile()* - Methode um.

Leider lief auch diese Entwicklung nicht ohne Probleme ab. Wir hatten die selben Probleme wie die bereits unter Rootkit-Control erwähnten, also im Wesentlichen die Kommunikation zwischen Userspace und Kernel. Das größte Problem, an dem auch einige der geplanten Features scheiterten, ist die Art und Weise wie im Kernelbereich programmiert werden muss. Jeder kleine Programmierfehler führt z.B. zum Absturz des gesamten Betriebssystems, weshalb wir für die Entwicklung schnell auf eine virtuelle Maschine umgestiegen sind. Weiterhin sind viele Standard-C-Methoden wie z.B. *strcp* oder *malloc* im Kernel nicht erlaubt und müssen durch sichere Varianten aus der Microsoft API ersetzt werden. Genau diese Methoden in den tiefen des MSDN (Microsoft Developer Network) oder der schlechten DDK-Dokumentation zu finden kostete uns einige Tage wertvoller Entwicklungszeit.

Deshalb war der komplizierteste Teil die Entwicklung des Whitelist-Vergleichs. Durch das Umleiten des *NtCreateFile()* - Systemcalls erhielten wir eine sehr große Anzahl von Events, da nun nicht nur die Aufrufe aller Applikationen, sondern auch Aufrufe von Windows selbst hier bei uns aufschlugen. Es entstanden also jede Menge irrelevante *CreateFile()* Events, die im Kernel gefiltert werden sollten. Deshalb entschieden wir uns für die Erstellung einer Whitelist und stuften nur Dateien, die nicht in dieser Liste enthalten sind, als kritisch ein. Hierfür mussten wir uns eine Reihe von Hilfsmethoden schreiben und ausgiebig in der virtuellen Maschine testen.

Schnittstellen

Für die Kommunikation mit dem UserSpace, also dem Rootkit-Control, verwenden wir die IOCTL-Schnittelle. Da diese nur aus dem Userspace aufgerufen wird, hat das Rootkit selbst keine Möglichkeit ein Event zu melden. Es muss also in regelmäßigen Abständen abgefragt werden.

Mögliche Erweiterungen

Der Whitelist-Vergleich basiert aktuell auf einer Reihe von einfachen String-Vergleichen, da die Implementierung der geplanten Regular-Expression-Funktionalität aus zeitlichen Gründen nicht mehr möglich war. Da hierdurch theoretisch die Möglichkeit besteht, unser Rootkit zu überlisten sollte diese Funktionalität nachgerüstet werden.

4.3.4 SOAP-Client

Kurzbeschreibung der Komponente

Der SOAP-Client ist das Modul des *BrowserControls*, welches Befehle zur Management-Konsole schickt und somit den aktuellen Status der einzelnen VM an den Benutzer übermittelt. Es ist somit die Schnittstelle zu den anderen Komponenten der Architektur. Wenn die *BrowserControl.exe* in einer virtuellen Maschine nun gestartet wird, sendet der SOAP-Client eine Nachricht an die Gegenstelle in der Management-Konsole und teilt dort mit der IP-Adresse mit, dass der Rechner bereit ist, eine URL zu überprüfen. Als Antwort wird eine noch nicht benutzte URL von der Management-Konsole gesendet. Nach der Überprüfung sendet unser SOAP-Client den Zustand der Webseite (und damit auch der virtuellen Maschine).

Eingesetzte Technologien & Verlauf der Entwicklung

Zuerst versuchten wir mit Apache AXIS für C++ [*http://ws.apache.org/axis/cpp/index.html]* die benötigten Stubs für die WSDL-Dateien zu generieren. Da AXIS ursprünglich eine Implementierung für Java-Stubs ist, unsere Applikation jedoch C++ verwendet, benutzten wir die AXIS – C++ Komponente. Die Stubs liessen sich generieren, allerdings konnten wir kein Testprogramm, bei dem wir die Stubs und die benötigten AXIS-Bibliotheken einbanden, erfolgreich kompilieren oder danach linken.

Vor allem aufgrund des Zeitmangels und der Tatsache, dass dies die letzte noch nicht funktionierende Komponente in unserer Entwicklungsphase war, beschlossen wir, uns nach einer Alternative zu AXIS C++ umzuschauen.

Nach kurze Suche stießen wir auf gSOAP, einem Framework welches der AXIS – Idee sehr ähnelt [*http://gsoap2.sourceforge.net]*. Die Generierung der Stub-Klassen war ebenso verhältnismäßig einfach. Zwar war die Kompilierung unseres Testprogrammes auch dort nicht sehr einfach, einige Linkeroptionen mussten durprobiert werden, um das gSOAP – Framework einzubinden, dennoch gelang dies uns nach einigen Anläufen.

Als Nächstes gliederten wir dieses Modul in unsere Hauptapplikation des *BrowserControls* ein. Neben ein paar weiteren Hürden wie das Auslesen der IP-Adresse unter Windows konnten wir jedoch erste Erfolge verzeichnen, die uns den Transport eines URL Strings über SOAP ermöglichten. Jedoch mussten wir für jeden Kommunikationsaufruf (getNextURL()) eine einzelne Methode schreiben. Schliesslich funktionierte aber auch dieser letzte Baustein für unsere komplette Architektur.

Schnittstellen

Das SOAP-Client-Modul selbst ist die Brücke zwischen der *BrowserControl-*

Applikation und dem SOAP-Gegenpol an der *Management-Konsole*. Das gSOAP-Framework hat dies auf beiden Seiten (C++ - Klassen für das BrowserControl und die SOAP-Kommunikation auf der Management-Konsole) erfolgreich lösen können.

4.3.5 IE-Remoting

Beschreibung

Das *IE-Remoting*-Modul stellt die Komponente zur Steuerung des Internet Explorers dar. Es ist ein Programm, das den Internet Explorer aufruft und es ermöglicht, den Browser von anderen Maschinen aus fernzusteuern. Mithilfe des *BrowserControl*-Moduls sollen dann die zu prüfenden angesurft werden.

Das *IE-Remoting*-Modul bietet die gesamte Funktionalität des Microsoft Web-Browsers und ist in der Lage, auf Prozessinformationen und den Prozessstatus einer Internet Explorer - Instanz zuzugreifen.

Anforderungen der Komponente

- Internet Explorer starten und beenden
- Explorer überwachen (Ladestatus)
- URL-Handling (In den Explorer laden und auslesen)
- Steuerung des Internet Explorers („vor", „zurück" etc.)
- Prozess-Handling (PID, Prozess-Status)
- Externe Ansteuerung des Internet Explorers
- Kommunikation mit dem Rootkit (RPC, IPC)
- Kommunikation mit dem Management-Server (XML, SOAP)

Eingesetzte Technologien

Zu Beginn wurden mehrere Technologien, die in den Kontext „Web-Browser Steuerung" fielen, betrachtet und ausgewertet. Folgende Technologien/Tools kamen in die nähere Auswahl:

- WINTASK (Browser testing tool)
- JACOBIE (Java based API for the use with Internet Explorer)
- JSOOP (Java Script object oriented, cross-browser JavaScript library)
- INSITE (.NET)

- JIFFIE (Java Interface for Internet Explorer)
- MFC (Microsoft Foundation Classes)

Das IE-Remoting-Modul wurde letztendlich hauptsächlich in der Sprache C / C++ und den Microsoft Foundation Classes implementiert.

Die Microsoft Foundation Classes (MFC) sind eine Sammlung objektorientierter Klassenbibliotheken, die von Microsoft für die Windowsprogrammierung und windowsbasierten Anwendungen mit C++ entwickelt wurden.

Die MFC dienen als Schnittstelle zu den nicht objektorientierten API-Funktionen des Betriebssystems und vereinfachen den Umgang mit den vom Betriebssystem zur Verfügung gestellten Ressourcen erheblich.

Unser Programm selbst ist objektorientiert geschrieben, als Entwicklungsumgebung diente Visual Studio 2005. Anfangs entwickelten wir mit dem Eclipse SDK 3.1, es stellte sich aber schnell heraus, dass Visual Studio 2005 besseren Entwicklungssupport beinhaltet - deshalb fiel die Entscheidung schnell auf das Microsoft-Entwicklungstool.

Verlauf der Entwicklung

Zu Beginn untersuchten wir das Aufgabengebiet, um mögliche Lösungen für unsere Anforderungen zu finden. Die ersten Brainstormings und Internetrecherchen ergaben eine Vielzahl an Vorgehensweisen und Tools. Nach einiger Zeit der Einarbeitung und der Vertiefung des Themengebiets „Browser Control" enstanden die ersten Ansätze und Versuche, dieses Modul zu realisieren.

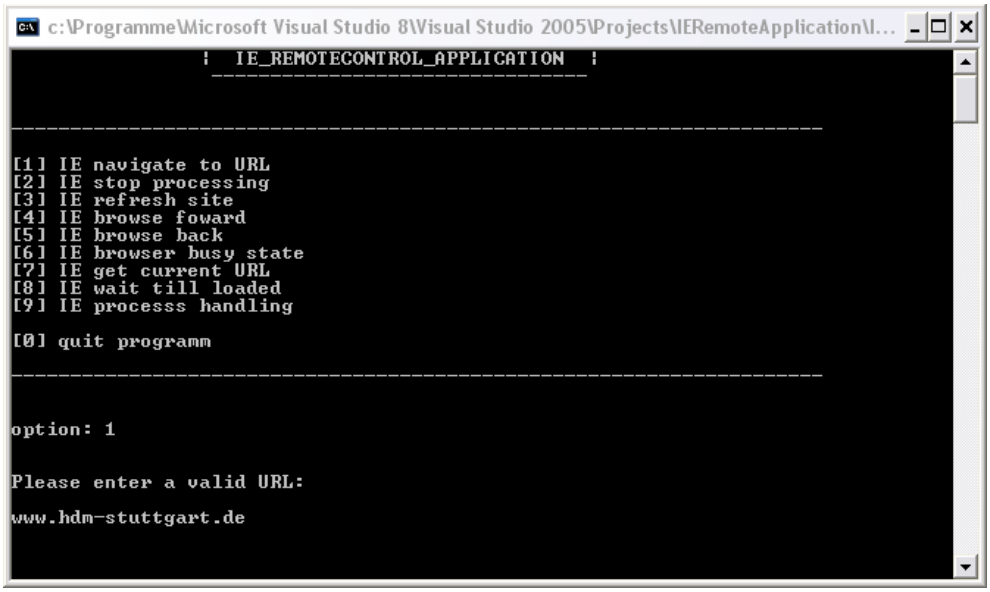
Leider hatten die meisten Tools einen zu großen Overhead und waren zu komplex, was es uns unmöglich machte, uns komplett in die Technologien einzuarbeiten.

Nach einigen Projektteam-Meetings musste zwischen C/C++, der nativen Programmiersprache des Betriebssystems und Visual Basic Script entschieden werde. Letztendlich einigten wir uns auf C/C++ MFC, da die Erfahrung mit C/C++ bereits vorhanden war.

Doch schnell wurde klar, dass C/C++ Erfahrung alleine nicht aussreichte, sich in der Windows-Programmierung zurechtzufinden. Durch Internet Recherche versuchten wir, Konventionen und Notationen der MFC herauszufinden.Das MSDN stellte hier keine sonderliche Hilfe dar.

Als erste Programmsteuerung entwickelten wir das Starten des Internet Explorers und die Steuerung des Browser.

Zum Testen schrieben wir eine kleine Menüsteuerung in C/C++ :



In weiteren Projektmeetings stellte sich heraus, dass es sinnvoll wäre den Prozessstatus und Prozessinformation des Internet Explorers zu erhalten. Dadurch wären wir in der Lage den Prozess per Befehl zu stoppen und zu steuern, oder zu überprüfen, ob ein Exploit den Internet Explorer selbst beendet haben könnte.

Bei der ersten Implementierung schlich sich ein Programmierfehler ein, den wir erst nach mühseligem Debugging entdeckten. Der Befehl um den Browser-Prozess zu schließen, beendete das komplette Betriebssysten. Es stellte sich heraus, dass wir nicht den Prozess sondern den „Parent Process" schließen wollten, welcher beim Internet Explorer eben Windows selbst ist.

Schnittstellen

Das Browser-Modul musste lediglich URLs vom SOAP-Client erhalten und an den Internet Explorer weiterleiten. Weitere Funktionalitäten zu anderen Modulen innerhalb der *BrowserControl*-Komponente hatten wir zunächst nicht geplant, jedoch ist dieses Modul selbst die Schnittstelle zwischen dem Browser und unserer *BrowserControl*-Applikation.

Mögliche Erweiterungen

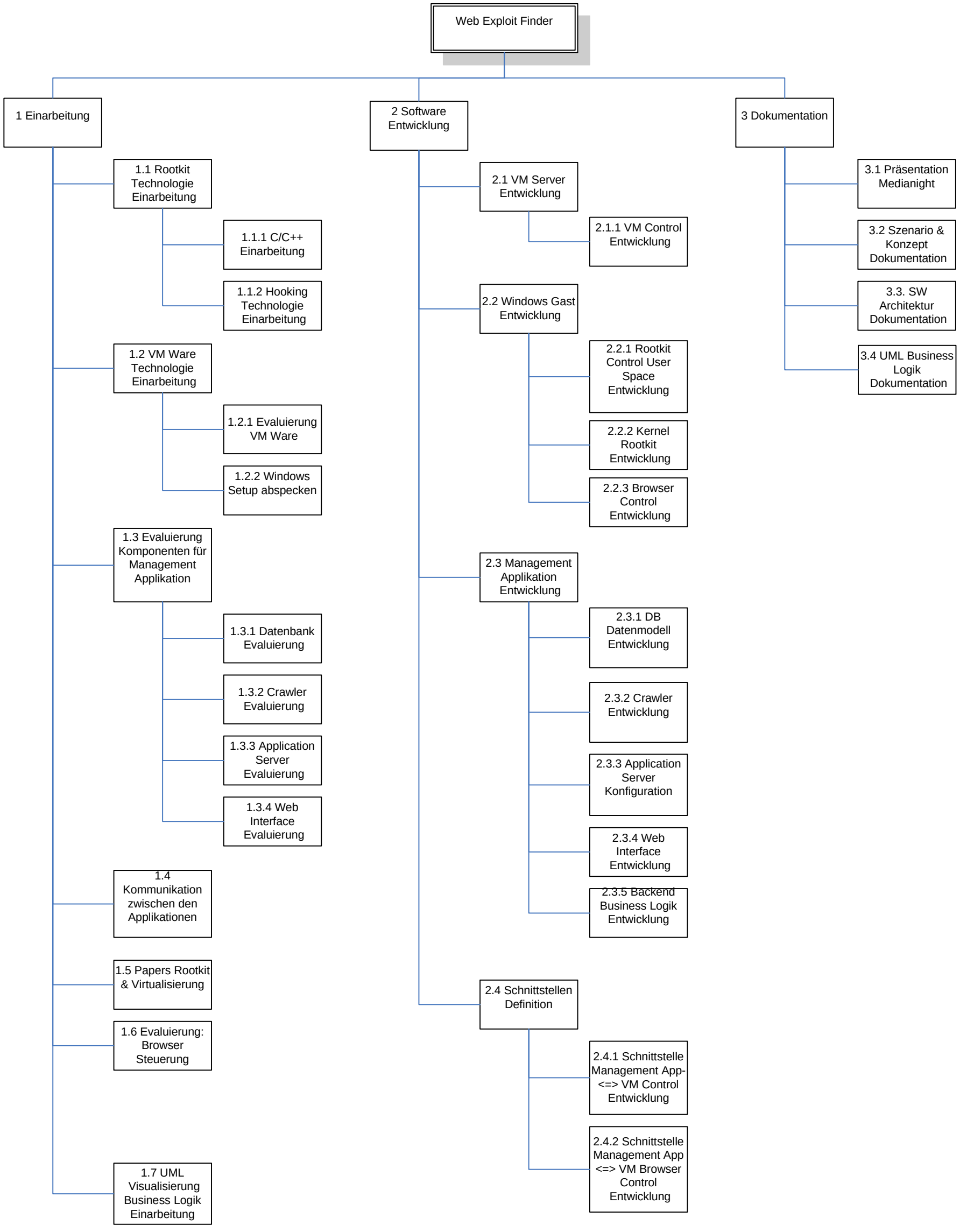
Unser System ist sehr modular aufgebaut, um einzelne Komponenten austauschen zu können. Von Beginn an war klar, dass wir dieses Konzept auch auf andere Browser wie beispielsweise Mozilla Firefox [http://www.getfirefox.com] ausweiten wollten. Dazu muss lediglich die Komponente ausgetauscht werden, die den Browser fernsteuert, um Firefox anzusprechen. Diese Erweiterung ist äußerst interessant, vor allem im Hinblick auf bestehende Exploits für Firefox, auf welche unsere Architektur anspricht [http://browserfun.blogspot.com/2006/07/mobb-28-mozilla-navigator-object.html], und die im verbreiteter werden.

5 Anhang

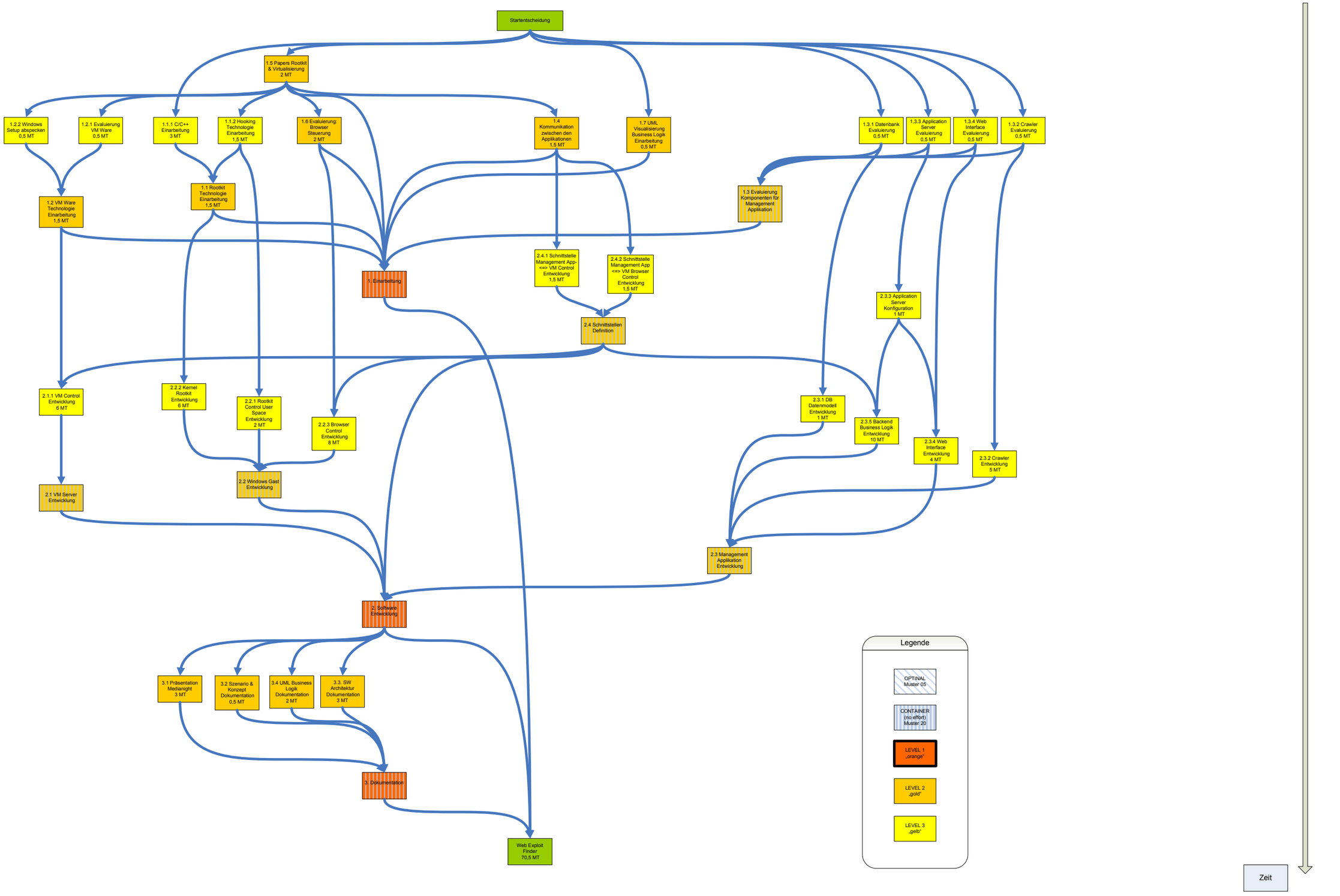
5.1 Projektstrukturplan

Produktstrukturplan: Web Exploit Finder V 1.4

Dienstag, 30. Mai 2006



5.2 Projektflussdiagramm


Startentscheidung
1.5 Papers Rootkit & Virtualisierung 2 MT
1.2.2 Windows Setup abspecken 0,5 MT
1.2.1 Evaluierung VM Ware 0,5 MT
1.1.1 C/C++ Einarbeitung 3 MT
1.1.2 Hooking Technologie Einarbeitung 1,5 MT
1.6 Evaluierung: Browser Steuerung 2 MT
1.4 Kommunikation zwischen den Applikationen 1,5 MT
1.7 UML Visualisierung Business Logik Einarbeitung 0,5 MT
1.3.1 Datenbank Evaluierung 0,5 MT
1.3.3 Application Server Evaluierung 0,5 MT
1.3.4 Web Interface Evaluierung 0,5 MT
1.3.2 Crawler Evaluierung 0,5 MT
1.2 VM Ware Technologie Einarbeitung 1,5 MT
1.1 Rootkit Technologie Einarbeitung 1,5 MT
1.3 Evaluierung Komponenten für Management Applikation
1. Einarbeitung
2.4.1 Schnittstelle Management App <=> VM Control Entwicklung 1,5 MT
2.4.2 Schnittstelle Management App <=> VM Browser Control Entwicklung 1,5 MT
2.3.3 Application Server Konfiguration 1 MT
2.4 Schnittstellen Definition
2.1.1 VM Control Entwicklung 6 MT
2.2.2 Kernel Rootkit Entwicklung 6 MT
2.2.1 Rootkit Control User Space Entwicklung 2 MT
2.2.3 Browser Control Entwicklung 8 MT
2.3.1 DB Datenmodell Entwicklung 1 MT
2.3.5 Backend Business Logik Entwicklung 10 MT
2.3.4 Web Interface Entwicklung 4 MT
2.3.2 Crawler Entwicklung 5 MT
2.2 Windows Gast Entwicklung
2.1 VM Server Entwicklung
2.3 Management Applikation Entwicklung
2. Software Entwicklung
3.1 Präsentation Medianight 3 MT
3.2 Szenario & Konzept Dokumentation 0,5 MT
3.4 UML Business Logik Dokumentation 2 MT
3.3. SW Architektur Dokumentation 3 MT
3. Dokumentation
Web Exploit Finder 70,5 MT
Legende
OPTINAL Muster 05
CONTAINER (no effort) Muster 20
LEVEL 1 „orange"
LEVEL 2 „gold"
LEVEL 3 „gelb"
Zeit