



Introduzione

Sono ormai molti anni che acronimi come **XSS** (Cross Site Scripting) e **CSRF** (Cross Site Request Forgery) sono entrate a far parte del vocabolario dei professionisti di sicurezza informatica per indicare quelle tecniche di attacco che sfruttano vulnerabilità di siti web dinamici che non impiegano un sufficiente controllo dell'input utente.

In contesti poco controllati è possibile inserire codice nelle richieste web ed ingannare il browser degli utenti che lo esegue come se fosse parte della pagina originale. Con questa tecnica risulta quindi possibile influenzare il comportamento delle applicazioni web e il loro aspetto.

Anche se lo sfruttamento di questo tipo di vulnerabilità può non apparire particolarmente pericoloso, **XSS** e **CSRF** sono tra le tecniche di attacco alle applicazioni web più pericolose.

Molti attacchi reali infatti ne sfruttano le potenzialità, in combinazione con altre tecniche, con effetti particolarmente negativi sulla sicurezza dei client.

Il presente documento nasce da una ricerca mirata a verificare la riproducibilità di attacchi di tipo **XSS** e **CSRF** in ambiti non web, per esempio nelle applicazioni **Desktop** ed in altre architetture. In questa analisi sono stati tenuti in considerazione molti possibili scenari e saranno presentati alcuni ambiti in cui potrebbe essere possibile sfruttare vulnerabilità di applicazioni per alterarne l'aspetto (realizzando attacchi di **phishing**) o il comportamento.

In una prima fase le vulnerabilità riscontrate sembravano non sfruttabili e comunemente con un livello minimo di pericolosità, tanto da risultare “trascurabili”.

Dallo studio di scenari più complessi, si è però potuto verificare che l'interazione tra più tecnologie, tipica delle attuali architetture informative, può rendere questa tecnica di attacco efficace e minare la sicurezza degli utenti.

Riteniamo questa documentazione assolutamente non esaustiva, anzi la proponiamo come prima analisi di tecniche di attacco che abbiamo definito “**Cross Application Scripting**” e “**Cross Application Request Forgery**”.

La stesura di questo documento è stata preceduta dalla verifica dell'esistenza di note tecniche o ricerche in ambito “**security**” e “**development**”.

Non sembra essere stata prodotta documentazione su questi temi.

Ciò lascia immaginare che le considerazioni e gli esempi prodotti non siano che la “punta di un iceberg” da segnalare con tempestività.

Concetto di Cross Application Scripting (CAS)

Similmente alle interfacce web, i moderni framework per la realizzazione di applicazioni grafiche (in questo documento si fa riferimento nello specifico a **GTK** e **QT**, i più importanti frameworks multiplatforma) permettono l'uso di tag all'interno di molti dei propri widgets.

Ciò implica la possibilità di formattazione particolarmente raffinate per il testo contenuto negli oggetti di tipo testo e la capacità di rappresentazione e gestione di contenuti multimediali (immagini, audio e video) o interattivi (links).

E' naturale che il proliferare del numero di funzionalità, se non gestite in modo corretto ed adeguato, possa rendere possibili utilizzi indesiderati della tecnologia, come la manipolazione della GUI (Graphical User Interface).

Esattamente lo stesso fenomeno che si realizza con l'uso di XSS in una pagina web.

E' proprio per questo motivo che abbiamo deciso di definire questo comportamento **CAS** (Cross Application Scripting).

Tipicamente le applicazioni desktop ricevono quantità considerevoli di input e supportano un numero elevato di features, sicuramente maggiori di qualunque interfaccia web.

Ciò rende più complesso per lo sviluppatore il controllo che tutti i dati provenienti da fonti insicure vengano filtrati correttamente.

Software vulnerabili a forme di **Cross Application Scripting** di base sono molti, incluse numerose applicazioni appartenenti a produttori noti.

Concetto di Cross Application Request Forgery (CARF)

Come evidenziato per il **Cross Application Scripting**, anche il **CARF** (Cross Application Request Forgery) è una riproduzione dell'analoga vulnerabilità web **CSRF** nelle applicazioni desktop.

Nel caso di **CARF** il concetto di “link” e di “protocollo”, ereditato dall'ambito web, è estremamente più esteso, dato che coinvolge componenti dell'ambiente grafico e, in alcuni casi, direttamente del sistema operativo.

Lo sfruttamento della vulnerabilità riconducibili a **CSRF** richiede una certa interazione da parte dell'utente.

Tale problematica in molti casi non è particolarmente vincolante poiché gli utenti possono essere facilmente indotti ad eseguire determinate azioni se l'interfaccia grafica del programma risulta opportunamente alterata.

Come detto, infatti, modifiche ingannevoli nell'aspetto delle applicazioni sono ottenibili con l'uso di **CAS**.

In questi contesti si può quindi parlare di una nuova modalità di “phishing”, la cui pericolosità è amplificata dalla mancanza di strumenti per la rilevazione di questo tipo di attacchi fuori dall'ambito web o di posta elettronica.

Al contrario delle tecniche di **XSS** che possono manipolare ed arrivare ad impartire comandi lato browser utente, attraverso **CAS** si può arrivare anche a dialogare con il sistema operativo e non solo con la sua interfaccia grafica.

Cosa sono le GTK e dove vengono utilizzate.

GIMP ToolKit (GTK) è un insieme di strumenti (toolkit) per la creazione di interfacce grafiche il cui principale strumento è la libreria definita “libgtk”. Il tool è sviluppato in linguaggio C e supporta nativamente l'ambiente grafico X Window System e Microsoft Windows.

GTK è rilasciato come software libero (licenza GNU GPL) e le proprie librerie sono utilizzate da aziende commerciali, anche molto note, per numerosi prodotti di ogni tipologia.

L'utilizzo principale di **GTK** rimane comunque l'ambito open source e questo toolkit è parte fondamentale dell'ambiente desktop **GNOME** (usato da circa il 45% degli utenti GNU/Linux).

Rispetto alle librerie **QT** (che verranno analizzate in seguito) le **GTK**, per la loro minore complessità, presentano meno possibilità di manipolazione “maliziosa”.

Le vulnerabilità rilevate e presentate in questo documento si basano su uno dei componenti più rilevanti di **GTK** “Pango Markup” il quale consente di interpretare testo e renderlo graficamente in base alla descrizione realizzata attraverso l'uso di uno specifico insieme di tag. Questo semplice linguaggio è molto simile ad **HTML** (basato su **XML**) e permette l'inserimento di attributi per la formattazione del testo in quasi tutti i widget grafici.

Alterazione base delle GUI in GTK

I tag presenti nel modulo **Graphics.UI.Gtk.Pango.Markup** sono, dal punto di vista delle funzionalità, molto limitati e non hanno caratteristiche che consentano all'utente interattività o l'inserimento di elementi multimediali.

Il tag maggiormente utilizzato è ``, che presenta numerosi attributi:

Attributi	Descrizione
font_desc	Una stringa che identifica i font usati (esempio "Sans Italic 12").
font_family	La famiglia dei font da utilizzare (come "normal", "sans", "serif" o "monospace").
size	La grandezza dei font espressa da un numero o una da una stringa predefinita ('xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large') o da una grandezza relativa ('smaller' o 'larger').
style	Lo stile del testo ('normal', 'oblique', o 'italic').
weight	L'intensità del testo come stringa ('ultralight', 'light', 'normal', 'bold', 'ultrabold', 'heavy') o numero.
foreground	Colore espresso come RGB ('#00FF00') o stringa ('red', 'white', ...).
background	Colore espresso come RGB ('#00FF00') o stringa ('red', 'white', ...).
underline	Stile di sottolineatura ('single', 'double', 'low', o 'none').
rise	Il posizionamento verticale del testo in decimi di "em". può essere negativo o positivo.
strikethrough	Sbarramento del testo ('true' o 'false').

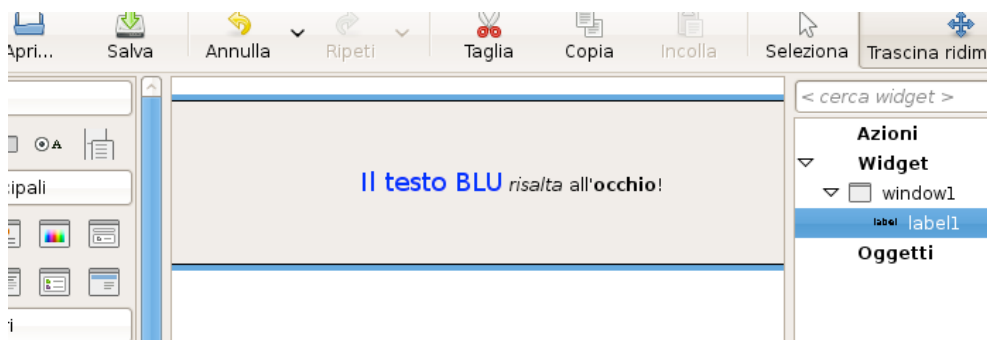
Altri tag rappresentano forme abbreviate di `` con determinati attributi:

Attributi	Descrizione
<code></code>	Bold
<code><big></code>	Equivalente a <code></code>
<code><i></code>	Italico
<code><s></code>	Strikethrough
<code><sub></code>	Subscript
<code><sup></code>	Superscript
<code><small></code>	Equivalente a <code></code>
<code><tt></code>	Monospace font
<code><u></code>	Underline

Un semplice esempio di stringa che utilizza il tag `` di GTK potrebbe essere:

```
"<span foreground="blue" size="x-large">Il testo BLU</span> <i>risalta</i>"
```

La rappresentazione grafica del codice è riportata nell'immagine successiva.



L'esempio riportato dimostra come sia possibile utilizzare questa tecnica di base per alterare l'aspetto delle applicazioni web. In determinati contesti questa preconditione è necessaria per minare la sicurezza.

L'unico prerequisito per il funzionamento di questo tipo di alterazioni è la mancanza di funzioni di “filtering” nell'applicazione bersaglio.

Per realizzare un attacco occorre quindi semplicemente individuare un campo di input di un'applicazione in cui sia possibile inserire il codice senza che questo sia filtrato. In assenza di controlli, in automatico, il codice inserito viene interpretato permettendo modifiche più o meno rilevanti nell'interfaccia grafica.

La pericolosità delle modifiche è strettamente legata all'applicazione e all'ambito di utilizzo. Per tali motivi possono verificarsi casi che non pregiudicano la sicurezza degli utenti e circostanze in cui può essere completamente minata.

Come ulteriore esempio di uso “innocuo” e particolarmente esplicativo di questa tecnica, utilizzeremo "Visualizzatore di Desktop Remoto" (GNOME 2.28).

Nel caso si cerchi di connettersi ad un host inesistente, il messaggio di errore dell'applicazione sarà simile a questo:



Nella precedente immagine si può notare come l'input dell'utente (*host.non.esisto*) viene presentato a video. I dati passati inizialmente dall'utente vengono riportati per intero all'interno del messaggio di errore. Ciò può essere un sintomo di un potenziale vulnerabilità a **Cross Application Scripting**.

Utilizzando una stringa di test si potrà confermare se il campo di inserimento viene filtrato o meno e completare così la rilevazione:

`TEST`

Inserendo questa stringa nell'applicazione come se fosse l'host a cui si intende connettersi sarà possibile avere conferma della vulnerabilità.



“Stranamente” nella finestra di dialogo che appare nell'applicazione non vi è traccia dei tag e il nome dell'host (**TEST**) è in grassetto. Abbiamo trovato un campo di input non filtrato ed il tag `` è stato interpretato: è un **Cross Application Scripting**.

Fin qui nulla di particolarmente interessante ma modificando opportunamente l'input si può creare un messaggio di errore plausibile che induca l'utente a compiere un'azione utile all'attaccante.

Analizzando meglio il primo screenshot, si può vedere che il testo tra virgolette (») è in corsivo. Ciò significa che l'applicazione utilizza, per default, il tag `<i>` per la formattazione del messaggio. Per generare un codice ad hoc per l'applicazione in questione è sufficiente inserire il nome dell'host chiudendo il tag `<i>` e le virgolette precedentemente proposte al fine di chiudere lo stile.

vnc.google.com</i>»

Nel tentativo di rendere il messaggio più credibile, a questa prima parte possiamo appendere un messaggio di errore, contenente il testo necessario a “suggerire” all'utente un'azione risolutiva:

vnc.google.com</i>» non è andata a buon fine. Si consiglia la disabilitazione

Perché la stringa risulti corretta pero' manca ancora qualcosa: bisogna gestire in qualche modo la corretta chiusura del tag <i> che l'applicazione apre in automatico. Se non lo facciamo il messaggio non sarà visualizzato poiché il codice XHTML complessivo non risulterà valido.

Arriviamo quindi alla stringa completa:

vnc.google.com</i>» non è andata a buon fine. Si consiglia la disabilitazione
b> del firewall per il corretto funzionamento del programma. La connessione
«<i>remota

Ed ecco lo screenshot che mostra il raggiungimento di un risultato utile ad un possibile attaccante:



Il messaggio di errore appare realistico e probabilmente l'utente può essere indotto a disabilitare effettivamente il proprio firewall.

L'attacco, nel caso dell'applicazione proposta, non ha molto senso perché non è possibile inserire il codice nel campo di testo in modo indiretto. L'esempio serve comunque ad avere un'idea di come sia possibile modificare la GUI attraverso del codice di markup. Questo approccio è alla base di tutte le successive sperimentazioni.

Implicazioni di sicurezza molto più importanti si verificano se l'applicazione attaccata fa uso della rete o di altri servizi, scenario che verrà analizzato nelle sezioni successive.

Cosa sono le QT e dove vengono utilizzate.

QT, come GTK, è un toolkit, ossia un'insieme di strumenti e di librerie per lo sviluppo di programmi dotati di interfaccia grafica. Si basa su widget (congegni o elementi grafici).

Le librerie Qt venivano inizialmente sviluppate dall'azienda Qt Software (meglio conosciuta come Trolltech), oggi proprietà di Nokia, che l'ha acquistata visto il notevole successo che le librerie riscuotono anche nello sviluppo di software commerciale.





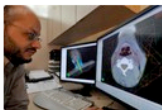



Ad oggi le librerie QT sono utilizzate per applicazioni quali Adobe Photoshop Album, Google Earth, KDE, Opera, OPIE, Skype, Qt Extended, VLC media player e VirtualBox.

Il toolkit è utilizzato per lo sviluppo di applicazioni in moltissimi ambienti, dai sistemi operativi per personal computer e server (**Windows, Linux, Mac Os X, *BSD, Solaris**) a **sistemi operativi embedded per cellulari o palmari**.

Nokia, nel proprio sito web (<http://qt.nokia.com/qt-in-use>), riporta un'ampia panoramica sui vari ambienti in cui vengono attualmente utilizzate le librerie:

Learn how Qt is Transforming your Industry

Discover how organizations like yours are gaining a competitive edge with Qt by developing next-generation user experiences including dynamic Web interaction.

 Qt in Automotive Infotainment	 Qt in Aerospace	 Qt in Home Media	 Qt in IP Communication
 Qt in Medical	 Qt in Oil & Gas	 Qt in Visual Effects	 Qt in MID's/Netbooks & Linux Distro's

Le features offerte dal framework QT sono molto più estese rispetto a quelle di GTK e permettono anche l'esecuzione di task molto eterogenei: dal Networking all'interazione coi Database, dal Parsing di formati quali XML e HTML al Multiprocessing.

Probabilmente è proprio la versatilità e ricchezza di funzionalità di questo framework che ne ha decretato il successo. Di contro, ovviamente, la complessità e l'estensione delle proprio API (Application Programming Interface) lo rende più facilmente sfruttabile da eventuali attaccanti di applicazioni e sistemi.

Le funzioni delle librerie QT supportano nativamente un loro linguaggio di markup, che è un subset piuttosto corposo dell'HTML:

Attributi	Descrizione
<a>	Link Supporta gli attributi href e name.
<address>	Address
	Grassetto
<big>	Big Font
<blockquote>	Paragrafo indentato
<body>	Document body
 	Line break
<center>	Paragrafo centrato
<cite>	Citazione inline
<code>	Stile codice inline
<div>	Document division, supporta gli attributi standard
<dl>	Definition list, supporta gli attributi standard
<dt>	Definition term, supporta gli attributi standard
	Enfasi

Attributi	Descrizione
	Font, supporta gli attributi standard
<h 1>/<h 2>/<h3>/..	Level Heading, supporta gli attributi standard
<head>	Document header
<hr>	Linea orizzontale, supporta gli attributi standard
<html>	HTML document
<i>	Italico
	Immagine, supporta gli attributi standard
<kbd>	User-entered text
<meta>	Meta-information, supporta gli attributi standard
	List item
<norb>	Non-breakable text
	Lista ordinata, supporta gli attributi standard
<p>	Paragrafo, supporta gli attributi standard
<pre>	Preformatted text
<qt>	Qt rich-text document, sinonimo per HTML
<s>	Strikethrough
<small>	Small font
	Grouped elements
	Strong
<sub>	Subscript

Attributi	Descrizione
<sup>	Superscript
<table>	Tabella, Supporta: border, bgcolor, cellpadding, cellspacing, width, and height.
<td>	Table data cell, supporta gli attributi standard
<th>	Table header cell, supporta gli attributi standard
<thead>	Table header, usato per le tabelle oltre una pagina
<title>	Titolo del documento
<tr>	Table row, supporta l'attributo bgcolor
<tt>	Typewrite font
<u>	Underlined
	Unordered list, supporta gli attributi standard

Alterazione base delle GUI in QT

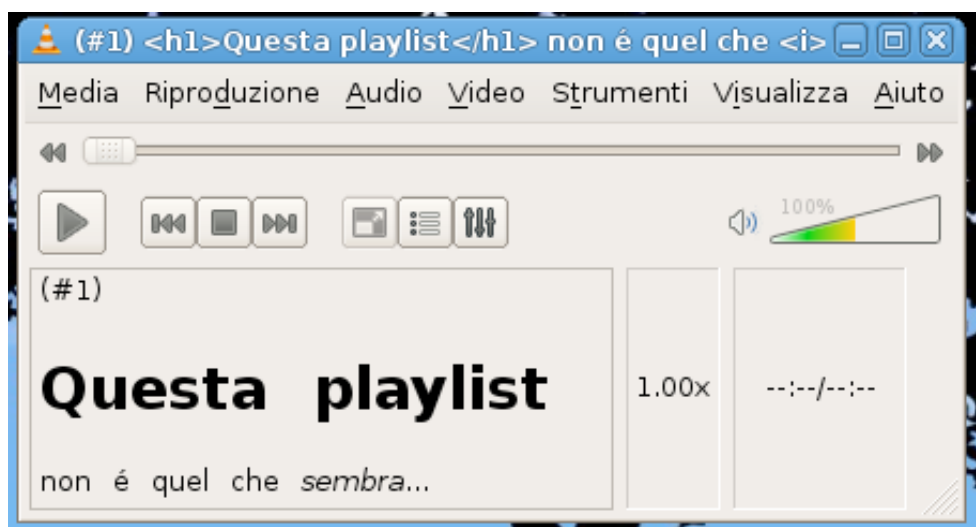
Come si è potuto vedere, i tag supportati dal framework QT sono veramente molti e anche per queste librerie analizzeremo alcune possibilità di attacco.

Anche in questo caso gli esempi presentati mostreranno un utilizzo di base e sono stati realizzati esclusivamente a scopo dimostrativo, come **Proof of Concept** della tecnica.

Evitando di ripetere i concetti fondamentali già espressi in precedenza, passiamo subito alla pratica.

Utilizzando **VLC**, un noto player multimediale opensource, abbiamo notato che l'applicazione non esegue adeguatamente il parsing dei file **playlist** (.pls) e quello dei tag di vari formati musicali. Ciò dimostra che questo player potrebbe essere utilizzato come vettore di attacco qualora gli fossero forniti, in input, sia file playlist che diversi formati di file audio opportunamente preparati. Le vulnerabilità sono quindi molteplici e dipendenti esclusivamente dal tipo di input fornito.

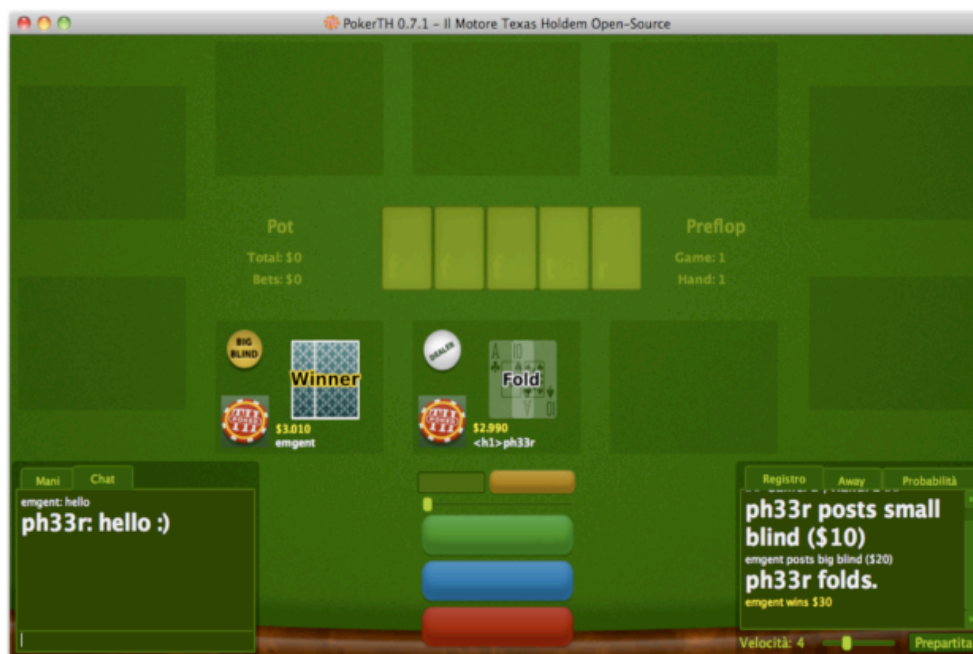
Nel primo esempio viene simulata l'attività di un utente che apra con vlc un file playlist, opportunamente predisposto da un attaccante. Come si può vedere dal codice sorgente poco sotto qualche alterazione consente di sfruttare le vulnerabilità del sistema di filtering dei file di input dell'applicazione:



Lo stesso comportamento si può ottenere modificando i **tag** di file **MP3** o **OGG Vorbis**.

Un ulteriore esempio di questo tipo di alterazione delle interfacce è stato realizzato nell'applicazione **PokerTH**, un popolare programma per il gioco del poker online.

L'applicazione è multiplatforma, disponibile per Windows, Linux e Mac Os X in forma precompilata o in formato sorgente.



Dallo screenshot si nota chiaramente che è stato effettuato un CAS nel nome del giocatore “ph33r”. In questo caso un qualunque giocatore è in grado di far apparire proprio codice, opportunamente formattato, agli utenti in gioco.

Questo esempio dimostra che è possibile applicare attacchi CAS contro utenti remoti, indipendentemente dal sistema operativo sul quale l'applicazione sia in esecuzione. Il sistema host in questo caso è Mac Os X.

Alterazione avanzata delle GUI in QT

Gli esempi finora presentati hanno presentato esclusivamente applicazione di base delle tecniche di CAS e CSRF. In questo paragrafo accenneremo ad alterazioni più evolute delle applicazioni che, alterando opportunamente le interfacce ed i comportamenti dei programmi, potrebbero consentire di realizzare dei veri e propri attacchi alla sicurezza di sistemi e dati degli utenti.

In questi scenari, i tag sfruttabili da eventuali attaccanti sono quelli che consentono all'utente di interagire con l'applicazione. In particolare vedremo degli esempi realizzati con i tag `<a>` e `` i quali, con i loro attributi, consentono di **interagire con l'esterno** dell'applicazione ed offrono la possibilità di combinare questo comportamento con altre tecniche per compiere veri e propri attacchi ai sistemi.

Gli attributi dei tag sono i seguenti:

Attributi	Descrizione
<code><a></code>	href (link), name
<code></code>	src (sorgente dell'immagine), source (nel caso di Qt3), width, height

Gli scenari in cui risulta sfruttabile il tag `` sono più complessi e sarà trattato in seguito. `<a>` è quasi sempre utilizzabile e consente di creare una sezione dell'interfaccia che risponda al click dell'utente, indirizzandolo verso la destinazione voluta dall'attaccante.

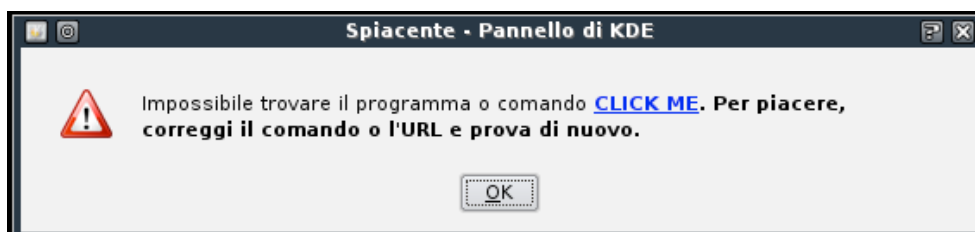
La prima applicazione di questa tecnica è sicuramente il **phishing**. Nel caso di attacco più semplice l'utente, credendo di collegarsi ad un sito noto ed affidabile, viene indirizzato su un dominio controllato dall'attaccante con tutte le conseguenze che ne derivano.

L'uso del tag `<a>` è banale: si specifica un **URI** attraverso l'attributo **href** e tutto ciò che è racchiuso dal tag diventa un hyperlink sensibile al click.

Per testare velocemente delle applicazioni e vedere se permettono questo tipo di interazione si può usare la seguente stringa di prova:

```
<a href="http://www.google.com">CLICK ME</a>
```

Nel caso di applicazione vulnerabile ad attacchi di tipo CSRF l'output sarà qualcosa del tipo:



Un'applicazione vulnerabile come quella appena presentata consente di implementare attacchi di **phishing applicativo**. Ne presentiamo ora un semplice (e peraltro innocuo) esempio, usando la console per passare la stringa di codice. L'applicazione che viene sfruttata in questo esempio è **gwenview**, il visualizzatore di immagini di KDE 4.

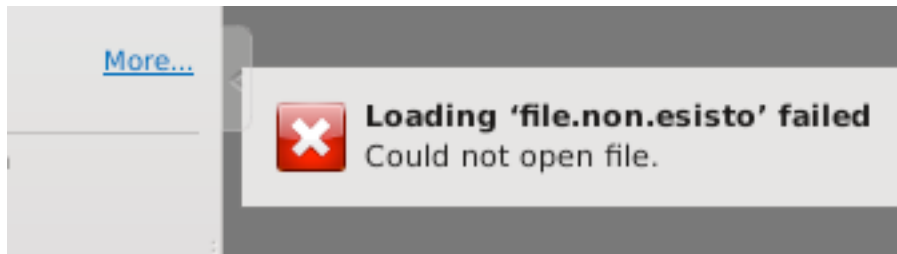
Il comando di cui sarà richiesta l'esecuzione è il seguente:

```
gwenview '<font face="verdana">libraries failed.<br>You can download missing libraries  
<a href="www.google.com">here.<font size="
```

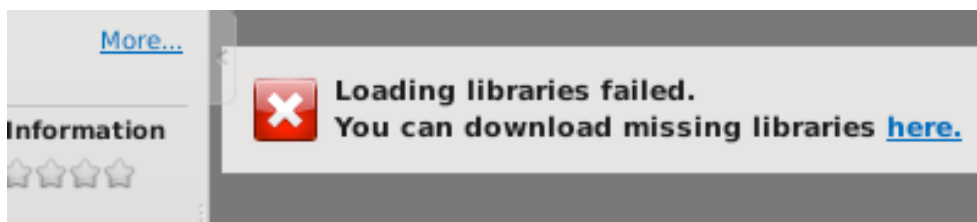
Si noti il tag incompleto posto alla fine. Questo è uno dei possibili modi per eliminare eventuale testo scritto dopo la stringa di attacco.

Negli screenshot che seguono è possibile vedere come un messaggio di errore di **gwenview** possa cambiare dopo una opportuna manipolazione.

Messaggio di errore normale:



Messaggio di errore manipolato:



Il testo evidenziato in blu (“here”) è un **link** che, cliccato, apre una finestra del browser di default verso la pagina indicata dall'attaccante.

Tecniche di Embedding

Un altro scenario estremamente interessante è rappresentato dalle applicazioni che utilizzano un **browser web integrato** o addirittura un **layout engine** per il rendering delle pagine web. Quest'ultima possibilità è stata realizzata nelle Qt 4.4, dove è stato integrato il motore di rendering **WebKit**.

Moltissime applicazioni già utilizzano questa feature, la cui popolarità è in costante aumento.

Per il prossimo esempio è stato utilizzato **Akregator** 1.2.9, il componente del Desktop Environment KDE per la lettura di feed RSS/Atom.

Il programma sfrutta le librerie KDE per il rendering delle news. L'applicazione non consente l'interpretazione di tutti i tag, in particolar modo ignora i tag che consentono di inserire comandi di scripting (ad esempio codice javascript), per limitare potenziali attacchi.

I test che sono stati eseguiti hanno permesso di scoprire, nonostante le protezioni adottate, rimane possibile l'inserimento di codice anche se in modo indiretto, attraverso determinati campi di un file **OPML**.

OPML (Outline Processor Markup Language) è un formato XML utilizzato per una rappresentazione strutturata e gerarchica dei contenuti e, in questo caso, consente di importare o esportare una struttura di feed preesistente.

Questo file OPML è stato modificato per analizzare il comportamento di **Akregator**:

```
<?xml version="1.0" encoding="UTF-8"?>
<opml version="1.0" >
  <head>
    <text></text>
  </head>
  <body>
    <outline isOpen="true" id="1" text="&lt;img src=&quot;http://www.google.it/intl/it_it/images/logo.gif&quot;>&lt;br>&lt;h1>Testo in H1&lt;/h1>&lt;br>&lt;a href=&quot;http://www.google.com&quot;>Link a google.com&lt;/a>" >
    </outline>
  </body>
</opml>
```

Il codice inserito in questo contesto non presenta differenze rispetto a quello utilizzato precedentemente, a parte il necessario encoding di alcuni caratteri per rendere il file XML-valid.

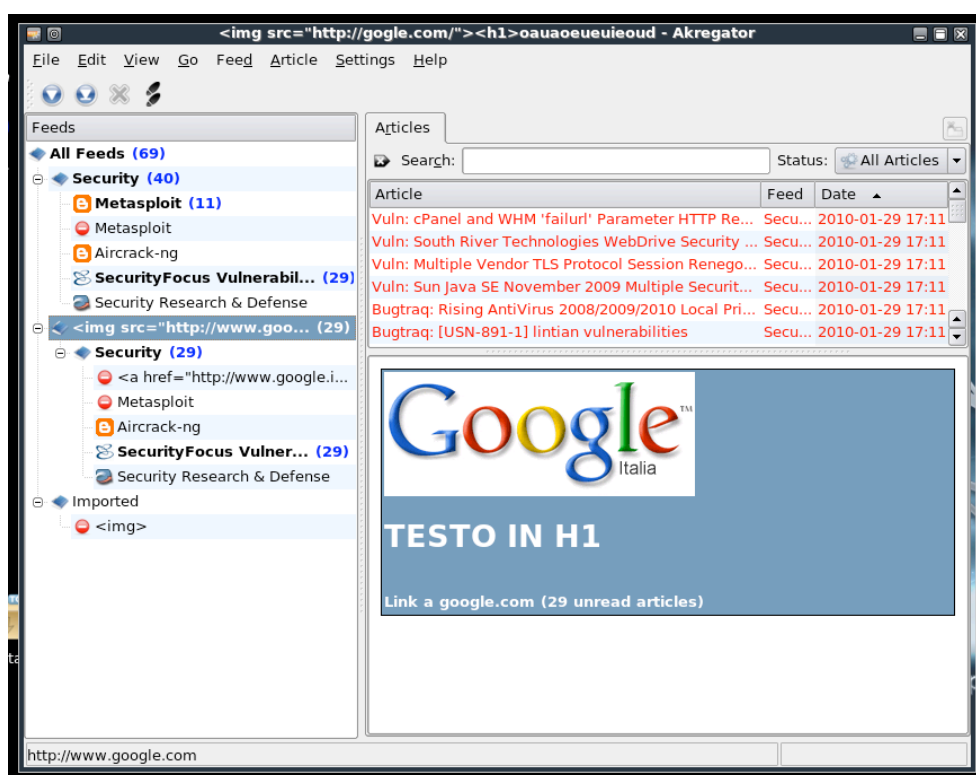
Il test contenuto nel file è il seguente:

```
<outline isOpen="true" id="1463565480" text="&lt;img src=&quot;http://www.google.it/intl/it_it/images/logo.gif&quot;>&lt;br>&lt;h1>Testo in H1&lt;/h1>&lt;br>&lt;a href=&quot;http://www.google.com&quot;>Link a google.com&lt;/a>" >
```

In questo caso, viene sfruttato il nome di un raggruppamento di feeds per forzare l'interpretazione di questo codice:

```
<br><h1>Testo in H1</h1><br><a href="http://www.google.com">Link a google.com</a>
```

Come è possibile vedere dallo screenshot successivo, il rendering html integrato permette il caricamento di immagini remote e l'utilizzo di tutti i tag visti in precedenza.



Il vantaggio nell'utilizzare `` rispetto al tag `<a>` consiste nel fatto che il contenuto presente all'indirizzo indicato nella proprietà `src` è caricato immediatamente e non richiede interazione con l'utente.

Questo comportamento può essere sfruttato utilizzando elementi esterni all'applicazione, come vedremo in seguito.

QT Scripting module

QtScript é un motore di scripting presente nel toolkit Qt dalla versione 4.3.0. Il linguaggio di scripting é basato su **ECMAScript** con delle estensioni caratteristiche delle Qt.

Uno scripting engine è una funzionalità estremamente potente perché consente di **utilizzare le API** delle librerie, direttamente da script, abbattendo enormemente i tempi di sviluppo e permettendo alle applicazioni di supportare facilmente plugin ed estensioni.

Per accedere a classi o oggetti c++ all'interno degli script, queste vanno esportate esplicitamente dal programma. Gli sviluppatori di Nokia sono molto attivi nello sviluppo di questo supporto ed hanno recentemente rilasciato una versione piuttosto stabile di un progetto per la generazione automatica dei bindings: **Qt Script Generator**.

Il noto player musicale **Amarok 2** è già compatibile con queste features e consente l'estensione dell'applicazione via plugin aggiuntivi.

Un esempio molto semplice di utilizzo del modulo è riportato nella documentazione:

```
QPushButton button;
QScriptValue scriptButton = myEngine.newQObject(&button);
myEngine.globalObject().setProperty("button", scriptButton);

myEngine.evaluate("button.checkable = true");

qDebug() << scriptButton.property("checkable").toBoolean();
scriptButton.property("show").call(); // call the show() slot
```

Nell'estratto di codice si può notare la creazione dell'oggetto QPushButton e la propria esportazione all'interno dello script. Questa procedura consente il completo accesso all'oggetto via script e permette di richiamarne metodi o impostarne proprietà.

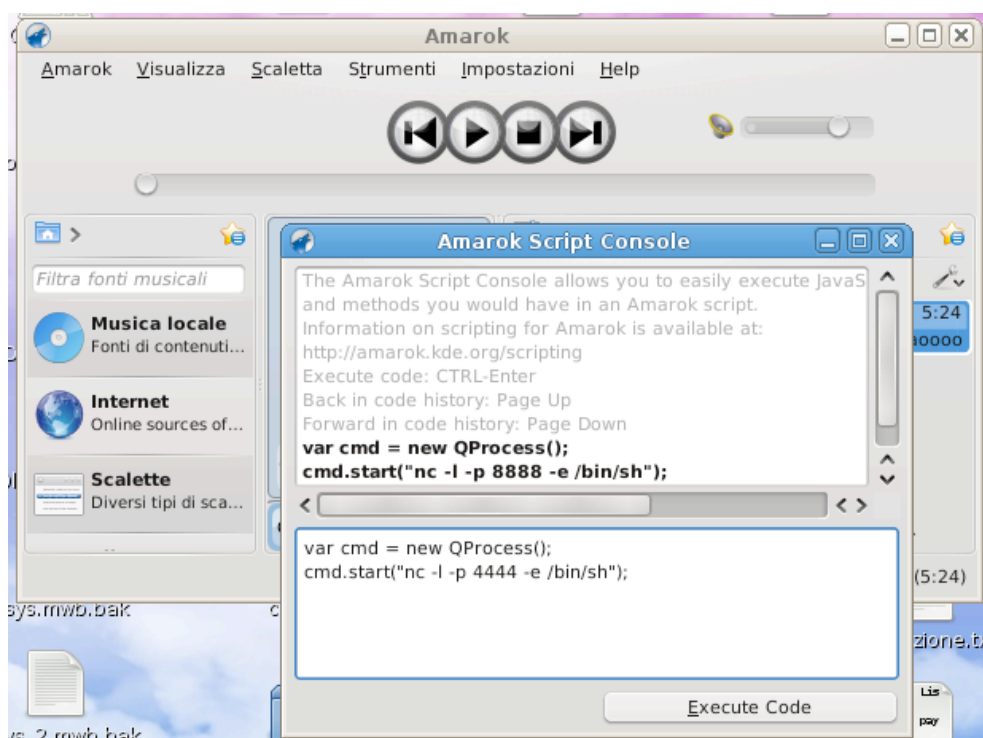
Il codice è naturalmente eseguito a runtime e può essere modificato dinamicamente.

Nonostante attualmente non sia molti diffusi programmi che utilizzino in modo scorretto questo modulo, teniamo a rimarcare le preoccupanti conseguenze sulla sicurezza che questo utilizzo produrrebbe.

Viste le configurazioni di default di QT, sarebbe possibile **iniettare nelle applicazioni** dello “shellcode” (codice ECMAScript) che avrebbe accesso a molte più risorse rispetto a shellcode tradizionali.

Un codice simile a quello presentato di seguito, in determinati scenari, potrebbe aprire una backdoor sul sistema bersaglio:

```
var cmd = new QProcess();  
cmd.start("nc -l -p 4444 -e /bin/sh");
```



Fortunatamente, almeno nel caso di Amarok2, tutti i test eseguiti con questo tipo di codice non hanno presentato vulnerabilità sfruttabili. Non è ancora chiaro se la sicurezza è garantita dalla robustezza del codice di Amarok2 o se sono particolarmente validi i meccanismi di protezione degli oggetti cui QTScrip ha accesso.

L'analisi di QTScrip dovrà essere di certo approfondita visto che sembra non esistano ancora precedenti in questa tecnica di attacco. Ciò che appare assolutamente necessario è che gli sviluppatori prestino particolare attenzione ai meccanismi di esportazione degli oggetti (**riducendo** al minimo le **classi esportate**) e che si presti particolare attenzione al **filtering degli input utente**.

Focus Introduzione

In questo documento si sono analizzati alcuni casi in cui è possibile sfruttare le tecniche CAS e CARE.

Solitamente la sfruttabilità di queste tecniche deriva da una programmazione non particolarmente attenta da parte delle aziende o dei singoli programmatori che si appoggiano alle librerie di terzi.

In alcuni casi non è così e in particolar modo nelle librerie QT, estremamente complesse, il comportamento di default di alcune classi è vulnerabile.

Un esempio di “anomalia” interna alle librerie QT (molto diffusa, ma che non permette attacchi dannosi) è il comportamento della classe QFileDialog, utilizzata nell'80% delle applicazioni QT per selezionare files da aprire o salvare.

Focus Qfiledialog

La documentazione delle librerie, nella sezione specifica di questa classe, propone il seguente esempio:

"The easiest way to create a QFileDialog is to use the static functions. On Windows, Mac OS X, KDE and GNOME, these static functions will call the native file dialog when possible."

```
fileName = QFileDialog::getOpenFileName(this,  
tr("Open Image"), "/home/jana", tr("Image Files (*.png *.jpg *.bmp)"));
```

Effettivamente la stragrande maggioranza dei programmatori usa la funzione statica getOpenFileName() per la scelta di file da filesystem.

Il problema si trova all'interno del file src/gui/dialogs/qfiledialog.cpp (le righe variano da versione a versione delle librerie, in ogni caso è possibile trovare lo snippet di codice cercando la stringa per il preprocessore "#ifndef QT_NO_MESSAGEBOX").

Il codice responsabile dell'anomalia è questo:

```
if (!info.exists()) {
    #ifndef QT_NO_MESSAGEBOX
        QString message = tr("\nFile not found.\nPlease verify the "
                               "correct file name was given");
        QPushButton *button = d->buttonBox->button(acceptMode() == AcceptOpen ?
        QDialogButtonBox::Open : QDialogButtonBox::Save);
        QMessageBox::warning(this, button->text(), info.fileName() +
        message);
    #endif // QT_NO_MESSAGEBOX
}
```

Come è possibile notare il messaggio è creato attraverso una semplice concatenazione di stringhe:

```
"info.fileName() + message"
```

Nella maggior parte dei casi è questa modalità di gestione dei messaggi d'errore che permette l'utilizzo di tecniche **CAS** e **CARF**. Una buona pratica è un utilizzo più accorto delle funzioni di concatenazione, anche nel caso non siano note tecniche d'attacco per la particolare tecnologia utilizzata.

Porre rimedio al bug di `QFileDialog` è piuttosto semplice. Il blocco di codice che segue è un breve esempio che permette di aggirare la generazione automatica del messaggio di warning:

```
void secureFileDialog()
{
    QStringList fileNames;
    QFileDialog fileDialog(this, tr("Open Image"), "/home/", tr
    ("Image Files (*.png *.jpg *.bmp)"));

    fileDialog.exec();
    fileNames = fileDialog.selectedFiles();
}
```

Non si utilizzano funzioni statiche ma viene istanziato un oggetto con gli attributi corretti per quanto riguarda la working directory, il filtro per le estensioni dei files, e il titolo del dialog. Successivamente viene poi richiamato il metodo `exec()` e raccolti i file selezionati attraverso `selectedFiles()`.

I files non subiscono controlli di sorta, in modo da non permettere messaggi di errore automatici. Il controllo deve essere fatto dall'applicazione, anche se reso facile dalla classe `Qfile`:

```
QFile checkFile(fileName);

// check if file exists
fileexists = checkFile.exists();
if (fileexists)
{
    ...
}
```

Focus Anteprima Files

Un altro problema di `QFileDialog` è la feature che permette l'anteprima dei files. Anche in questo caso il mancato controllo dell'input può implicare l'interpretazione di eventuali tag maliziosi nel nome del file.

L'unica soluzione trovata è quella di disabilitare l'anteprima del file disabilitando la seguente flag (valida per QT3):

```
FileDialog::setPreviewMode(QFileDialog::NoPreview);
```

Stiamo conducendo ulteriori ricerche al riguardo, ed è probabile che ci saranno ulteriori pubblicazioni e approfondimenti nei prossimi mesi riguardo i comportamenti dei framework grafici ed eventuali problematiche interne agli stessi.

Naturalmente, utilizzando una politica di responsabile disclosure, ogni azienda o ente coinvolto verrà adeguatamente informato delle problematiche riscontrate.

KIOslaves e protocols di KDE

KDE (**K Desktop Environment**) è un ambiente grafico desktop per postazioni di lavoro Unix. L'ambiente è basato sulle librerie **QT** e funziona sulla maggior parte dei sistemi operativi BSD, Microsoft Windows e OS X.

I programmi e le librerie che compongono KDE sono rilasciati sotto licenze GPL ed LGPL. Al contrario degli altri ambienti grafici, KDE, si distingue per la sua nota facilità d'uso e per la sua intuitività e documentazione. Apparentemente molto simile ad un sistema grafico Microsoft Windows è forse l'ambiente desktop più utilizzato.

Le **KIOslave** (**KDE Input Output slave**.) sono dei piccoli programmi che consentono l'accesso diretto a filesystem locali, server, ftp e via scorrendo tramite un'unica **API**.

Le applicazioni scritte utilizzando **KIO** possono lavorare su file salvati all'interno di server remoti esattamente come se fossero sul computer locale, in maniera trasparente all'utente e indipendentemente dal protocollo utilizzato per il trasferimento dati.

L'infrastruttura KIO fornisce una libreria per le applicazioni che si occupa di tenere traccia dei protocolli utilizzabili e di avviare, se necessario, il programma appropriato — detto KIOslave o ioslave — per la gestione del protocollo di comunicazione richiesto.

L'applicazione che sfrutta l'infrastruttura KIO non deve far altro che specificare un URI che indichi il protocollo da usare; se questo è disponibile, l'operazione di I/O avrà luogo. La libreria KIO codifica e decodifica automaticamente i dati ed invia notifiche usando i segnali di sistema di Qt.

Di seguito è riportato un elenco delle KIOslaves più utilizzate:

KIOslaves	Utilizzo
file:/	Permette la lettura di file e la loro esecuzione

KIOslaves	Utilizzo
data:/	Permette di includere piccoli documenti o codice.
apt:/	Permette di dare comandi diretti al gestore dei pacchetti APT.
fish:/	Permette di accedere a demoni OpenSSH
nfs:/	Permette l'accesso a risorse NFS
tar:/, zip:/, gzip:/, bzip:/, bzip2:/	Permette la navigazione dentro file compressi
applications:/	Permette l'accesso alla configurazione del sistema
finger:/	Permette di conoscere se l'host remoto ha servizi finger attivi.
settings:/	Permette l'accesso alla configurazione del sistema
smb:/	Permette l'accesso alle risorse condivise
sftp:/	Permette il trasferimento sicuro dei file via SSH
desktop:/	Mostra il contenuto della cartella Desktop
cgi:/	Permette di eseguire script cgi senza webserver
telnet:/	Permette il login remoto via telnet.
rlogin:/	Permette di stabilire sessioni con server che supportano rlogin.

Conclusioni

Le tecniche sopra descritte sono da considerarsi esclusivamente una potenziale “base” per attacchi combinati con i quali e’ possibile minare la sicurezza degli utenti. La pericolosità e l'impatto dell'uso di queste tecniche sono dipendenti esclusivamente dalle applicazioni attaccate e dalle funzioni per le quali sono realizzate.

Sono stati svolti con successo test su piattaforme: **Microsoft Windows**, **Mac OS X**, **GNU/Linux** (KDE e GNOME), *BSD e **Symbian OS**.

Per comprendere la reale pericolosità di questa tecnica è sufficiente pensare ad un tentativo di attacco che tenti di indurre l’utente a svolgere semplici azioni. A tal fine possono essere soggette ad attacchi di tipo **CAS** molte delle applicazioni in grado di ricevere input da remoto (lettori feed rss, email client, file google earth, flussi streaming). L'utente potrebbe essere convinto, per esempio, ad accettare una Java applet, magari generata con **SET** (Social Engineering Toolkit) ed apparentemente firmata in modo corretto.

I vari scenari e le molteplici applicazioni della tecnica, come già accennato precedentemente, variano anche in base al tipo di piattaforma.

Nel caso di device mobili e smartphone, per esempio, la pericolosità e l’impatto di un attacco **Cross Application Scripting** o **Cross Application Request Forgery** sono particolarmente elevati, data la struttura stessa dei sistemi. Nella maggior parte dei casi infatti in tali dispositivi i programmi sono eseguiti con accesso privilegiato dal solo utente del sistema.

Negli ambienti KDE based, invece, la tecnica trova forse il bersaglio più vulnerabile visto il supporto alla maggior parte delle features dei sistemi. Apparentemente i limiti alle modalità d'attacco sono dettati esclusivamente dall'inventiva dell'attaccante nell'uso di KIOslaves. Anche in questo ambiente possono essere integrate tecniche di attacco locali e remote.

Realizzazioni pratiche di attacchi combo basati su queste tecniche non sono stati appositamente presentati all’interno di questo documento per rispettare una policy di divulgazione di tipo “responsible disclosure” con i vendor delle applicazioni coinvolte.

Ci riserviamo comunque di pubblicare i risultati della sperimentazione in corso in un apposito futuro documento.

Crediti

Emanuele `emgent` Gentili

Emanuele Gentili è consulente e partner della società Gerix.IT, lavora nel campo della sicurezza informatica e della ricerca.

Socio dell' Associazione Informatici Professionisti, del Linux User Group di Orvieto e membro di OWASP si occupa prevalentemente di condurre e supervisionare Tiger Team nelle attività di Penetration Test e Vulnerability Assessment. Trainer partner di Offensive Security, si occupa di formazione live in lingua italiana per i master di certificazione Offensive Security.

Coordinatore del progetto BackTrack Linux, cofondatore di Exploit Data Base, security developer della distribuzione GNU/Linux Ubuntu. E' membro del consiglio di BackTrack Italia per la quale si occupa di ricerca e sviluppo.

<http://www.backtrack.it/~emgent/>

<http://twitter.com/emgent>

Emanuele `crossbower` Acri

Emanuele Acri collabora con Gerix.IT come sviluppatore di Jesys (The Web Intrusion Prevention System) e security researcher.

Dal 2009 collabora attivamente con gli sviluppatori BackTrack Linux ed e' attualmente parte del team di sviluppo.

Membro della community BackTrack italiana, ha lavorato alla sicurezza del server del progetto Exploit DataBase.

E' autore di diversi tools, tra cui la suite Complemento, e main developer di altri (Gerix-Wifi-Cracker-NG).

<http://www.backtrack.it/~crossbower/>

http://twitter.com/crossbower_bt

Alessandro `scox` Scoscia

Alessandro Scoscia svolge attività di consulenza come Internet Project Manager e come Web Developer Consultant.

E' membro dell'Associazione Informatici Professionisti e di Internet Webmaster Association.

E' business partner della società Gerix.IT per i servizi connessi alla sicurezza informatica e si occupa di ricerca security oriented.

Co-ideatore di Jesys (Jesys Effectively Saves Your Server) un Web Intrusion Prevention System.

E' membro del Consiglio di BackTrack Italia dove si occupa di ricerca e sviluppano ed uno dei "primi soci" del Linux User Group Orvieto.

<http://www.backtrack.it/~scox>

http://www.twitter.com/a_scoscia

Sitografia

http://it.wikipedia.org/wiki/Cross-site_scripting

http://it.wikipedia.org/wiki/Cross-site_request_forgery

<http://it.wikipedia.org/wiki/GTK%2B>

<http://www.gtk.org/api/2.6/pango/PangoMarkupFormat.html>

<http://www.haskell.org/gtk2hs/docs/current/Graphics-UI-Gtk-Pango-Markup.html>

http://it.wikipedia.org/wiki/Qt_%28toolkit%29

<http://doc.trolltech.com/qq/qq02-fun-fast-and-flexible-qt-script.html>

<http://doc.trolltech.com/4.6/scripting.html>

http://amarok.kde.org/wiki/Development/Scripting_HowTo_2.0

<http://doc.trolltech.com/4.1/qt.html>

<http://doc.trolltech.com/4.6/qfiledialog.html>

<http://doc.trolltech.com/4.6/qmessagebox.html>

http://en.opensuse.org/Konqueror_Tips_and_Tricks

http://it.wikipedia.org/wiki/KDE_Input_Output

<http://developer.kde.org/documentation/design/kde/ioslaves/index.html>

<http://doc.trolltech.com/4.3/richtext-html-subset.html>