

Goal-Oriented Web Search

by

Victor Lamont Williamson

B.S., Massachusetts Institute of Technology (2005)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science


at the

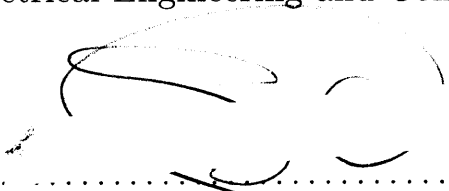
ARCHIVES

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

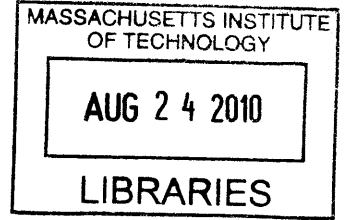
February 2010

© Massachusetts Institute of Technology 2010. All rights reserved.


 Author
 Department of Electrical Engineering and Computer Science
 Feb. 2, 2010


 Certified by
 Stephen A. Ward
 Professor
 Thesis Supervisor


 Accepted by
 Dr. Christopher J. Terman
 Chairman, Department Committee on Graduate Theses



Goal-Oriented Web Search

by

Victor Lamont Williamson

Submitted to the Department of Electrical Engineering and Computer Science
on Feb. 2, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We have designed and implemented a Goal-oriented Web application to search videos, images, and news by querying YouTube, Truveo, Google and Yahoo search services. The Planner module decomposes functionality in Goals and Techniques. Goals declare searches for specific types of content and Techniques query the various Web services. We choose which Web service has the best rating at runtime and return the winning results. Users weight their preferred Web services and declare a repository of their own Techniques to upload and execute.

Thesis Supervisor: Stephen A. Ward
Title: Professor

Acknowledgments

Special thanks to Prof. Stephen Ward from MIT's Laboratory for Computer Science and Artificial Intelligence for guidance and refinement of the project and for the opportunity to pursue my idea. No less thanks to PhD student Justin Mazzola Paluska for tweaking the Planner to suit the needs of the Web application and for technical support, design insights, and practical common sense.

Table of Contents

1	Introduction	13
1.1	The Web	14
1.2	Background	15
1.2.1	The Planner	15
1.2.2	Subgoals and Goal Trees	16
2	Implementation	17
2.1	Architecture	17
2.2	Goal Specifications	18
2.3	Repositories	18
2.4	Goals and Techniques	19
2.4.1	Video Search	21
2.4.2	Image Search	24
2.4.3	News Search	24
2.4.4	Mixed Search	26
2.5	Search Results	27
2.6	AJAX	28
2.7	Running Goals	29
2.8	Goal Trees	32
2.9	Heuristics and Biasing	33
2.10	Adding Techniques	35
2.10.1	Uploading	36
2.10.2	Debugging	36

2.11 Saving Settings	36
2.11.1 Cookies	36
2.12 HTML Display	37
2.12.1 Dynamic Layout	37
2.12.2 Home Page	38
2.12.3 Results Page	38
2.12.4 Cascading Style Sheets	39
3 Security Issues	41
3.1 Threat Model	41
3.2 Running Untrusted Code	42
3.2.1 Sandboxing	42
3.2.2 Closures	43
3.2.3 Tokens	44
3.2.4 Other Python Solutions	45
4 Alternative Designs	47
5 Future Work	49
6 Related Work	53
7 Conclusion	55

List of Figures

2-1	Architectural Design	18
2-2	Default Repository	20
2-3	Video Search Goal Spec	21
2-4	A Single Video Result - Truveo	23
2-5	A Single Video Result - Google	24
2-6	Image Search Goal Spec	25
2-7	Image Search Result - Yahoo BOSS	25
2-8	News Search Goal Spec	26
2-9	Single Video XML Result	27
2-10	Single Image XML Result	27
2-11	Single News XML Result	28
2-12	Mixed Search Module	29
2-13	MixVidImgNews Technique Module	30
2-14	Run Goal Code	32
2-15	Mixed Search Goal Tree	33
2-16	Web Goal Tree	34
2-17	Home Page	39
2-18	Video Search Results	40
3-1	Python Closures	44

List of Tables

2.1	Goals, Techniques and Web Services	19
3.1	Python Opcode Token Accesses	46

Chapter 1

Introduction

We have implemented a Goal-oriented Web search application to search online for videos, images, and news. The search results come from YouTube, Truveo, Google, and Yahoo Web Services. Users drag a slider to weight their preferred Web services. The home page includes a checkbox for users to choose which kinds of content to search and a text box allows users to enter a URL pointing to an external repository that declares other search implementations.

The application uses the Planner [5], a framework written in Python that allows new search implementations to be dynamically added at runtime. The Planner was built for applications in pervasive computing environments, and has been customized to integrate with Web search. The Planner leverages that pervasive applications are extensible and decomposable. The Planner separates Goal declaration from Goal execution so that new modules called Techniques can be plugged into an existing application without requiring code changes to the application. This approach has the nice property that decision logic chooses how to satisfy a Goal at runtime.

The outline of this paper is as follows. We show how the Web serves as motivation for Goal-driven search and give an overview of the Planner. In chapter 2 we discuss our software architecture and implementation. In chapter 3 we discuss security issues with uploading untrusted code and suggest remedial measures. In chapter 4 we discuss some alternative designs, and finally in chapters 5 and 6 we discuss future and related work.

1.1 The Web

The Web is a distributed platform on which we build our Web application. We combine multiple independent Web applications into a single application and compare their performance. Independently developed Web applications often implement the same or similar functionality and Web users evaluate them to determine which best suit their needs. Many providers, for example, have implemented Web search engines including Google, Yahoo, Live Search, Ask.com, Cuil and Bing. Web searchers either stick with a single search engine or alternate between various ones. Factors that influence user choices include site popularity, search speed, look-and-feel, and personal bias. Our Web application leverages ratings returned from Web search services to evaluate the quality of search results.

The Web has remotely accessible public APIs for enterprises and independent developers to build useful and novel applications. Web services integrate other Web services to produce powerful and feature-rich Web applications. Fast and robust feature rich applications were once restricted to desktop applications where they leverage the host operating system and CPU, while the Web was best suited for static markup transfer. Applications such as instant messaging, text editing, email, Web conferencing, and document management are now ported to the Web. Browser technologies such as AJAX and dynamic script loading have improved the look-and-feel of Web applications so that they compete with desktop applications.

One popular set of Web applications are called mashups. Mashups aggregate or “mash up” many different services on the Web to perform useful function. For example, the Google Maps API has been incorporated into mashups to show jobs, people, cab routes, photos, stores and disease outbreak information on a map. Mashups depend on organizations such as the World Wide Web Consortium (W3C) to standardize Web APIs to increase portability and ease application development. Many popular Web APIs use XML or JSON as the data interchange along with standard protocols such as REST, XML-RPC and SOAP. These standardizations allow applications running on any machine connected to the Internet to easily integrate with

and leverage Web services.

1.2 Background

We use the Planner [5] to allow for dynamic uploading and execution of third party Web search implementations. Our application operates differently than previous pervasive computing applications because the Planner is rerun on each Web request. We want end users to evaluate, compare and modify search results from different Web services. Our goal is to seamlessly integrate Planner capabilities within the Web browser and maintain application look-and-fell and ease-of-use.

1.2.1 The Planner

The Planner was designed for ubiquitous and pervasive computing environments where many and diverse computing platforms are often removed, relocated, upgraded, or replaced. Applications operating in ubiquitous environments are expected to stay abreast of environment changes. In order to achieve runtime determination of high-level implementation decisions the Planner relies on Goals and Techniques. Goals describe the high-level problem to be solved and prescribe decision points where the Planner chooses a particular Technique to satisfy a Goal. Techniques implement Goals and return domain-specific attributes for the Planner to compare alternative approaches.

Goal descriptions include Goal parameters, output attributes, documentation, and a URL that points to the Goal module containing evaluation code. Goal parameters restrict application semantics. For example, “play any movie” may be parameterized to “play a particular movie.” Goal attributes describe qualities of the implementation returned by Techniques. The evaluation formula maps attributes to a single value and instructs the Planner how to choose between competing Techniques. Goal documentation provides an overview and useful information for Technique developers.

Techniques are small scripts broken up into a series of declarative and arbitrary imperative code called stages that include subgoal declarations, evaluation code, and

commit code. Technique evaluation code is idempotent to handle being rerun when the environment changes or when the user makes new requests. The Planner executes each stage sequentially. Techniques may declare many subgoal and evaluation stages to incrementally estimate and refine Goal attributes. Development is distributed for independent and decentralized application development.

1.2.2 Subgoals and Goal Trees

Techniques may declare required subgoals to hierarchically decompose application functionality and to declare application-specific decision points. The Planner explores Goals and produces a Goal tree by binding Techniques to Goals and recursively binding Technique subgoals. The Goal tree is an and-or-tree and encapsulates all known implementations. A path from the root Goal node to a Technique leaf represents a single implementation strategy. The Planner heuristically chooses for each Goal the Technique that maximizes the value returned from the Goal's evaluation formula. The application may commit the current plan, change parameters, or upgrade a previously committed plan to accommodate removed, modified, or added system resources.

Chapter 2

Implementation

Python modules form the basis of the Planner implementation and development framework. Goals and Techniques are encapsulated as Python modules. Python is an interpretive language useful for both scripting and application development. We use Python to receive HTTP requests, to display HTML, and to easily incorporate and integrate Planner modules. We use Python print statements to generate HTML and JSON response data.

2.1 Architecture

The Web application consists of two python modules, one to display the home page and the second to run Goals. Figure 2-1 shows how the varying components interact at runtime. For each request, the Planner checks the default repository for declared Goals and Techniques. The Goal specification provides the necessary data to create fields on the home page and references the URL from which to download the Goal module. Each Technique module is downloaded and the home page is updated with the Technique's display name and Web services. On each Web search, the browser sends a request to run the chosen Goal using the Javascript XMLHttpRequest object and a JSON Goal tree is returned. The attributes are parsed and sent to the server using an HTML form. The server returns the results in HTML for display.

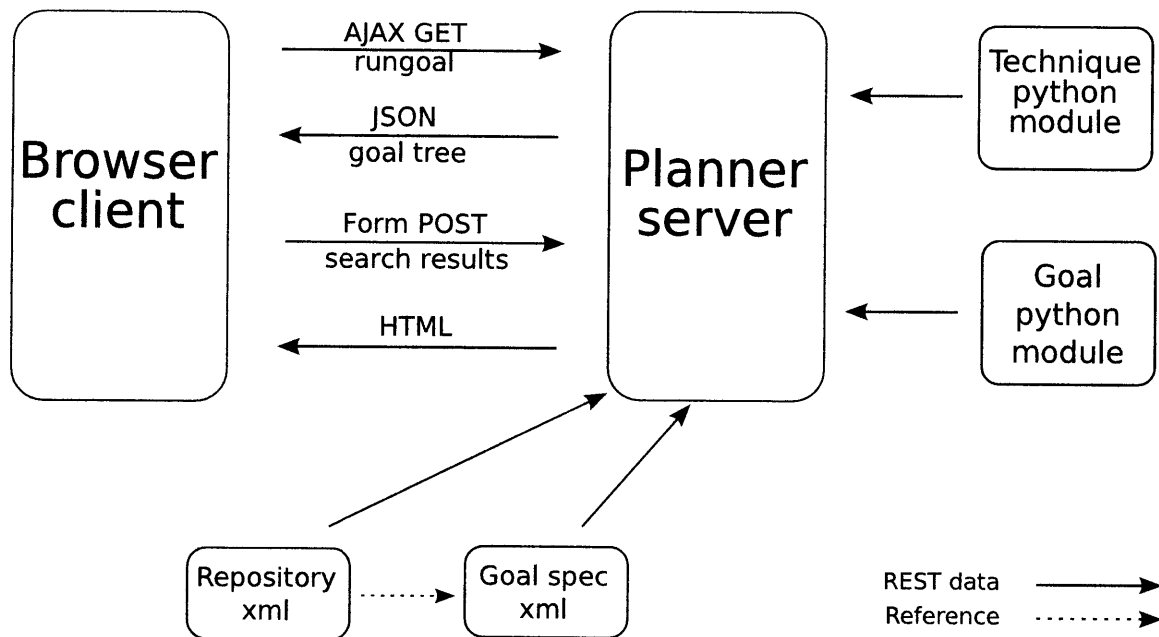


Figure 2-1: Architectural Design

2.2 Goal Specifications

Goals are specified in XML files that record the Goal name, display name, description, parameters, attributes, and HTML file for the Goal. The XML specification is a Web accessible document that the Planner accesses at runtime to populate the home page, download Goal modules, discover Techniques and generate HTML. Attributes encapsulate the data that associated Techniques must return. We use attributes for heuristic evaluation and to output domain-specific data in HTML. Techniques may return an *html* attribute with a link to an HTML display page, otherwise the Planner will use the site linked to by *default* in the Goal specification. This allows Techniques to customize how they display results in the browser. As example, section 2.4.1 shows the Video Search Goal specification.

2.3 Repositories

Repositories are a way to centralize and easily extend which Goals and Techniques are available to the Web application. We include a default repository that contains all of our Goals and Techniques. The repositories are formatted in XML and are

parsed using Python's *xml.dom.minidom*. Figure 2-2 shows our default repository. Each *goal* element references the corresponding Goal specification within its *href* attribute. Each *technique* element specifies the Goal it implements, a URL to its implementation and a list of sources corresponding to Web services it searches. We have a second repository to demonstrate user upload of additional Techniques. The second repository follows the same format and includes a single Technique to implement video search using the Google AJAX API. We use the *urlopen* method of Python's *urllib* module to read in the repository and parse it into a Document Object Model (DOM) object.

2.4 Goals and Techniques

We have created four Goals to organize search for online content. We have a Goal for each type of content and a Mixed Search Goal that includes the other Goals as subgoals. Table 2.1 summarizes our Goals, Techniques and queried Web services.

Goal	Technique	Web Service
Search Videos	YouTubeSearch	YouTube Data API
	AOLVideoSearch	Truveo XML API
	GoogleVideoSearch	Google AJAX API
Search Images	YahooImgSearch	Yahoo Build your Own Search Service (BOSS)
	GoogleImgSearch	Google AJAX API
Search News	YahooNewsSearch	Yahoo Build your Own Search Service (BOSS)
	GoogleNewsSearch	Google AJAX API

Table 2.1: Goals, Techniques and Web Services

```

<repository>
  <goal name="VideoSearch" href="http://people.csail.mit.edu/
    victorw/goals/videosearch.xml"/>
  <goal name="NewsSearch" href="http://people.csail.mit.edu/
    victorw/goals/newssearch.xml"/>
  <goal name="RateResults" href="http://people.csail.mit.edu/
    victorw/goals/rateresults.xml"/>
  <goal name="ImageSearch" href="http://people.csail.mit.edu/
    victorw/goals/imagesearch.xml"/>
  <goal name="MixedSearch" href="http://people.csail.mit.edu/
    victorw/goals/mixedsearch.xml"/>
  <technique name="YouTubeSearch" goal="VideoSearch" href="http://
    people.csail.mit.edu/victorw/techniques/youtubesearch.py">
    <source name="YouTube" href="http://gdata.youtube.com/
      feeds/api/videos"/>
  </technique>
  <technique name="YahooNewsSearch" goal="NewsSearch" href="http://
    people.csail.mit.edu/victorw/techniques/yahoonewssearch.py">
    <source name="Yahoo! News" href="http://search.yahooapis.com/
      NewsSearchService/V1/newsSearch"/>
  </technique>
  <technique name="GoogleNewsSearch" goal="NewsSearch" href="http://
    people.csail.mit.edu/victorw/techniques/googlnewssearch.py">
    <source name="Google News" href="http://ajax.googleapis.com/
      ajax/services/search/news"/>
  </technique>
  <technique name="AOLVideoSearch" goal="VideoSearch" href="http://
    people.csail.mit.edu/victorw/techniques/aolvideosearch.py">
    <source name="Truveo" href="http://xml.truveo.com/apiv3"/>
  </technique>
  <technique name="UserRate" goal="RateResults" href="http://
    people.csail.mit.edu/victorw/techniques/userrate.py"/>
  <technique name="RankRate" goal="RateResults" href="http://
    people.csail.mit.edu/victorw/techniques/rankrate.py"/>
  <technique name="YahooImgSearch" goal="ImageSearch" href="http://
    people.csail.mit.edu/victorw/techniques/yahooimgsearch.py">
    <source name="Yahoo! Images" href="http://search.yahooapis.com/
      ImageSearchService/V1/imageSearch"/>
  </technique>
  <technique name="GoogleImgSearch" goal="ImageSearch" href="http://
    people.csail.mit.edu/victorw/techniques/googleimgsearch.py">
    <source name="Google Images" href="http://ajax.googleapis.com/
      ajax/services/search/images"/>
  </technique>
  <technique name="MixVidImgNews" goal="MixedSearch" href="http://
    people.csail.mit.edu/victorw/techniques/mixvidimgnews.py"/>
</repository>

```

Figure 2-2: Default Repository

```

<goal name="VideoSearch">
  <summary>search and return web videos</summary>
  <display>search videos</display>

  <doc>Goal that searches the web for videos to be displayed
    to web viewers via the planner web application.
  </doc>
  <param name="text" type="string"/>

  <output>
    <attribute name="html" type="string" default="http://
      people.csail.mit.edu/victorw/planner/html/videosearch.py"/>
    <attribute name="videos" type="xml"/>
  </output>
  <evaluation href="http://people.csail.mit.edu/
    victorw/goals/videosearch.py"/>
</goal>

```

Figure 2-3: Video Search Goal Spec

2.4.1 Video Search

The Video Search Goal has three implementations for YouTube, Truveo and Google respectively. The Technique that searches Google is included in a separate repository from the default to demonstrate uploading a Technique. Figure 2-3 shows the Goal specification. The *text* parameter is the search text. The default HTML and evaluation code point to their respective Python module implementations.

The HTML results page includes the duration of the video, the title of the video and the view count. A tooltip containing a description of the video is triggered when the mouse hovers over the thumbnail. To prevent HTML parsing errors in tooltips we filter out control characters and only keep ASCII characters between 32 and 126. Quotations and HTML tags are removed to ensure tooltips are not susceptible to Cross-site Scripting (XSS) attacks. We use the Python *re* pattern matching module to filter out HTML tags and the *tooltip* method is placed in a utilities file for easy access across all files. The video title is contained within an HTML anchor tag and opens a new window where the video can be viewed. There is often insufficient space to display the full title and a tooltip on the anchor tag displays the full title.

We search YouTube, Truveo and Google AJAX API. The YouTube API allows us to search and upload videos, create playlists, customize players and playback, add YouTube widgets, track activity and more. We use the Data API to easily search videos by query string. We obtained a developer key and client ID and use

YouTube’s Python Client Library. We create a `YouTubeVideoQuery` object and pass it to a `YouTubeService` object which issues our search query. We set *max_results* to 15, include “restricted content” and order results by rating. We can access video properties by iterating over the entries returned in the video feed. We use the title, description, duration, view count, thumbnail and rating properties, but YouTube also exports the video’s publish date, category, tags, alternative formats, and geographic location.

We use AOL’s Truveo XML API to retrieve video search results from Truveo. Truveo also provides a JSON API and libraries for AJAX, Flash, and Ruby. We likely would have used JSON had it been available in 2008. There is no library support for the XML API. We use standard REST-style access methods over HTTP. We have an application id and use version 3 of the API. We pass “truveo.videos.getVideos” within the *method* parameter and max our results at 15. Truveo returns dozens of properties for its videos. Not all properties are normally populated. For example, we use the *userRating* property over Truveo’s proprietary *vRank* property to rate video results, but *userRating* is not always present, in which case we default to *vRank*. Figure 2-4 shows a single video result from Truveo.

Google Videos is searched using the Google AJAX Search API. Libraries are provided for Javascript and Flash. We do not use the libraries because our Techniques make requests from the server and not from the client. On version 1.0 of the API we set the *rsz* parameter to large and the query parameter *q* to the user’s search text. The Google AJAX API responds with the result set formatted in JSON which we pass to Python’s builtin *eval* function to create a Python dictionary. Before evaluation, we turn the response string into a Unicode literal by preceding it with *u’*. To prevent parsing errors, we remove ASCII character 127 and characters smaller than 32. Figure 2-5 shows a Google JSON response with search query “swahili”. The original source of this particular video entry is from YouTube. We use the *rating* property to rank Google’s search results.

```

<Video>
  <id>2061533051</id>
  <title>Klaus and Greta</title>
  <sponsored>0</sponsored>
  <videoUrl>http://xml.truveo.com/rd?i=2052820940&a=
    01bbddf413a052ff31d5a3a38b96078d&p=1&
    h=4b53d7e61d7881c:3101cd5b05806b67f69accb3d2c0f2f1
  </videoUrl>
  <channel>Hulu</channel>
  <channelUrl>http://www.hulu.com</channelUrl>
  <dateFound>Fri, 15 Jan 2010 05:55:25 -0500</dateFound>
  <textRelevancy>319</textRelevancy>
  <vRank>0.402201</vRank>
  <description>Jenna enters in a fake relationship with James Franco
    while Jack tries to gain access to a drunken voicemail
    he left Nancy.
  </description>
  <referrerPageUrl>http://www.hulu.com</referrerPageUrl>
  <copyright>2009 NBC Studios, Inc. All Rights Reserved</copyright>
  <rating>TV-14</rating>
  <runtime>1277</runtime>
  <category>Comedy</category>
  <tags>...</tags>
  <showName>30 Rock</showName>
  <showUrl>http://www.hulu.com/30-rock</showUrl>
  <episodeName>Klaus and Greta</episodeName>
  <episodeNumber>9</episodeNumber>
  <seasonNumber>4</seasonNumber>
  <country>United States</country>
  <language>English</language>
  <dateProduced>2010-01-09T01:51:05Z</dateProduced>
  <viewCount>38</viewCount>
  <thumbnailUrl> http://thumbnails.truveo.com/
    0011/09/DB/09DBBD49586EF0191329DB.jpg</thumbnailUrl>
  <videoResultEmbedTag>
    <embed src="http://xml.truveo.com/eb/i/2052820940/a/
      01bbddf413a052ff31d5a3a38b96078d/p/1/h/
      4b53d7e61d7881c:3101cd5b05806b67f69accb3d2c0f2f1"
      type="application/x-shockwave-flash"
      allowFullScreen="true"
      width=" 425" height=" 245"></embed>
  </videoResultEmbedTag>
  <redirectFor/>
  <adultFlag>0</adultFlag>
  <videoPlayerEmbedTag>...</videoPlayerEmbedTag>
  <formats>flash</formats>
</Video>

```

Figure 2-4: A Single Video Result - Truveo

```

{"GsearchResultClass":"GvideoSearch",
 "title":"Swahili Song (Regina)",
 "titleNoFormatting":"Swahili Song (Regina)",
 "published":"Fri, 26 May 2006 20:46:12 PDT",
 "content":"Swahili Song, East African, Kenya, Tanzania",
 "publisher":"www.youtube.com",
 "tbUrl":"http://3.gvt0.com/vi/AysrDPo3Vms/default.jpg",
 "tbWidth":"320",
 "tbHeight":"240",
 "videoType":"YouTube",
 "url":"http://www.google.com/url?q\u003dhttp://www.youtube.com/
 watch%3Fv%3DAysrDPo3Vms\u0026source\u003dvideo\u0026vgc\u
 u003drss\u0026usg\u003dAFQjCNF3UWvoYA3IIUYK0z2AnaoL72EMuA",
 "playUrl":"http://www.youtube.com/v/AysrDPo3Vms\u0026fs\u003d1\
 u0026source\u003duds\u0026autoplay\u003d1",
 "rating":"4.6860986",
 "duration":"246"}

```

Figure 2-5: A Single Video Result - Google

2.4.2 Image Search

Our application searches the Web for images. Figure 2-6 shows the Image Search Goal specification. We provide two implementations. One Technique searches the Yahoo Build your Own Search Service (BOSS) and the second searches the Google AJAX search API. The number of results returned from BOSS is limited to 15. Google limits its result set to 8 images.

Results pages include a thumbnail, a title and the size of the image. The full image can be viewed in a new tab by clicking on the title. A tooltip with the description is placed on the thumbnail and a tooltip with the full title is placed on the title's anchor element.

The Google AJAX search API searches images in very much the same way it searches videos described in section 2.4.1. For BOSS we obtain an application ID and limit our search to 24 results. The response is in XML. Figure 2-7 shows a single image result from BOSS.

2.4.3 News Search

The News Search Goal searches the Web for up-to-date news by query string. We search both Yahoo's Build your Own Search Service (BOSS) and the Google AJAX API for latest news. We use the same application ID that is used to search BOSS


```

<goal name="ImageSearch">
  <summary>search and return web images</summary>
  <display>search images</display>

  <doc>Goal that searches the web for images to be displayed
    to web viewers via the planner web application.
  </doc>
  <param name="text" type="string"/>

  <output>
    <attribute name="html" type="string" default="http://
      people.csail.mit.edu/victorw/planner/html/imagesearch.py"/>
    <attribute name="images" type="xml"/>
  </output>
  <evaluation href="http://people.csail.mit.edu/
    victorw/goals/imagesearch.py"/>
</goal>

```

Figure 2-6: Image Search Goal Spec

```

<Result>
  <Title>obama jpg</Title>
  <Summary>Edit This CALLING ALL LIBERAL DEMOCRATS What do
    you say we hold a Woodstock of our own Since those
    bitter old souls recently held what they described as
    a conservative Woodstock I m wondering
  </Summary>
  <Url>http://vickishipps.today.com/files/
    2009/09/obama.jpg</Url>
  <ClickUrl>http://vickishipps.today.com/files/
    2009/09/obama.jpg</ClickUrl>
  <RefererUrl>http://vickishipps.today.com/
    category/politics</RefererUrl>
  <FileSize>159232</FileSize>
  <FileFormat>jpeg</FileFormat>
  <Height>500</Height>
  <Width>350</Width>
  <Thumbnail>
    <Url>http://thm-a01.yimg.com/nimage/4d410097bef156bc</Url>
    <Height>145</Height>
    <Width>101</Width>
  </Thumbnail>
</Result>

```

Figure 2-7: Image Search Result - Yahoo BOSS

```

<goal name="NewsSearch">
  <summary>search and return the latest news articles from the web
  </summary>
  <display>search news</display>

  <doc>Goal that searches the web for news articles to be displayed
    to web viewers via the planner web application.
  </doc>

  <param name="text" type="string"/>

  <output>
    <attribute name="html" type="string" default="http://
      people.csail.mit.edu/victorw/planner/html/newssearch.py"/>
    <attribute name="news" type="xml"/>
  </output>
  <evaluation href="http://people.csail.mit.edu/
    victorw/goals/newssearch.py"/>
</goal>

```

Figure 2-8: News Search Goal Spec

for images. Figure 2-8 shows the News Search Goal specification. The specification references the link that produces HTML markup. The HTML displays the title within an anchor link, followed by a portion of the description, the publish date and the publisher. The full description can be viewed as a tooltip when the user's mouse hovers over the description area. The publish date lists the day, month and year using the *fromtimestamp* method of Python's *date* module. The publisher is within an anchor that links to the publisher's site. If one exists, a thumbnail is scaled down 20% and floated to the right of the description.

2.4.4 Mixed Search

The Mixed Search Goal combines search results for videos, images and news by declaring each of the corresponding Goals as subgoals. Using subgoals we demonstrate the Planner's ability to decompose functionality and allow users to combine content. The application passes a special 3-bit parameter called *types* to the Mixed Search Goal to specify which subgoals to run. The first bit instructs the Goal to search videos, the second to search images and the third to search images. Any combination of the bits may be set. At runtime, we use the Mixed Search Goal when two or more of videos, images, and news is searched.

There are two Techniques that implement the Mixed Search Goal. The first

```

<video>
  <title>Letterman - Betting on Tony Blair</title>
  <desc>Find out how an average citizen benefitted from the former
Prime MInister of England.</desc>
  <thumbnail>http://i.ytimg.com/vi/sAVD1046H8g/2.jpg</thumbnail>
  <viewcount>25588</viewcount>
  <duration>144</duration>
  <url>http://www.youtube.com/watch?v=sAVD1046H8g&feature=youtube_gdata</url>
</video>

```

Figure 2-9: Single Video XML Result

```

<video>
  <url>http://www.gwinnett.k12.ga.us/ArcadoES/Schoeller/
BlackHistory/mlk_index.jpg</url>
  <title>mlk index jpg</title>
  <desc>Encyclopedia of Prominent African Americans Click
on Picture to Hear Dr King s Famous Speech</desc>
  <thumbnail>http://thm-a01.yimg.com/nimage/
f01db0ec8f7f2b64</thumbnail>
  <height>258</height>
  <width>397</width>
</video>

```

Figure 2-10: Single Image XML Result

searches videos and images because of their similarity and the second Technique searches all content using the *types* parameter. We only use the second Technique. For combined video and image search specified by *types* = 3, we set the *html* attribute to a page that neatly displays only videos and images.

2.5 Search Results

We format search results as an XML attribute on the JSON response. To economize space we only return video, image, and news properties needed for HTML display, which always include the title, description and URL. Figure 2-9 shows the XML format for a single video result for search query “Tony Blair” where YouTube was the winning Technique. Figures 2-10 and 2-11 show image and article entries for search query “MLK”. In each case Yahoo was the winning Technique.

```

<article>
  <url>http://www.kens5.com/news/local/Hutchison-stops-in-
    San-Antonio-but-not-at-MLK-March-82000827.html</url>
  <title>Snubbed? Hutchison stops in S.A.
    but not at MLK March</title>
  <desc>Senator Kay Bailey Hutchison made a brief stop in San
    Antonio Monday morning but, unlike several other political
    candidates, she did not attend in the city's MLK March.
    "I was not particularly invited, so I didn't,
    she said. I would love to if invited consider
    that in the future." Hutchison says she acknowledges the
    importance of the day -- as well as the importance of all MLK
    ceremonies ...</desc>
  <published>1263918463</published>
  <modified>1263918464</modified>
  <source>KENS 5 San Antonio</source>
  <sourceurl>http://www.kens5.com/</sourceurl>
</article>

```

Figure 2-11: Single News XML Result

2.6 AJAX

We use Asynchronous Javascript and XML to run Goals on the server. As seen in figure 2-1 we run each Goal on the server using an AJAX request sent in the background and then direct the user's browser to the results page. We support both Internet Explorer and Firefox methods to create a request object. Firefox supports the World Wide Web Consortium (W3C) standard [10] to create an XMLHttpRequest object while Internet Explorer uses the *new ActiveXObject* syntax. We create a new request object for each request because we do not readily benefit from the performance advantages otherwise. Most users run no more than a few search queries before being directed to the search results, and requests are spaced far enough apart to forgo more sophisticated request queuing. For each *rungoal* request the XMLHttpRequest object's *onreadystatechange* callback is set. The callback displays the Goal tree, saves user settings, and directs the user to the HTML results page. We process the response after the request object has received all data as signified by *readystatechange=4* and verify HTML 200 status. The JSON response is trimmed by passing a literal regular expression to Javascript's *replace* function before being passed to *eval* and converted to a Javascript JSON object.

```

from planner.goal import Goal

class MixedSearch(Goal):
    """Goal to search items on the Internet"""

    def __init__(self, params=None):
        Goal.__init__(self, params, required_params={"text": "csail"})

    def evaluate(self, technique):
        return technique.toprating

```

Figure 2-12: Mixed Search Module

2.7 Running Goals

When the server receives an AJAX request, we kick off a *rungoal* script. The *rungoal* script creates a Goal object that is passed to the Planner object's *satisfy* method which returns a Plan object. We import the Goal, GTDatabase and Planner, read the default and user-specified repositories and download Goal and Technique modules into separate package directories. The *www* user is given write permissions on these directories using AFS's *fs setacl* command.

To execute Techniques and Goals we need to import them and add them to the GT-Database. The Web application is extensible and we do not presume what the Goals and Techniques are in advance. To evaluate and execute unknown strings at runtime we use Python's builtin *eval* method to evaluate expressions and Python's *exec* method to execute statements. For each Goal and Technique, we evaluate *add_goal* and *add_technique* to add them to the Planner database. We execute import statements for Goals and Techniques using Python's "from *module* import *class*" syntax. Each Goal and Technique module must have a class definition with the same name that is declared in the Goal specification. Figures 2-12 and 2-13 show the expected Goal and Technique interfaces. Each Goal must subclass the Goal class and each Technique must subclass the Technique class.

The Goal and Technique parent classes provide the machinery to evaluate Techniques and to separate Techniques into stages. The Goal class takes as input the Goal parameters and has an unimplemented *evaluate* method. The *technique* module includes three Python decorators to execute subgoal, evaluation and update stages.

```

class MixVidImgNews(Technique):

    GOAL = MixedSearch

    def __init__(self,goal):
        self.timescommitted = 0
        Technique.__init__(self,goal)

    @subgoal('sg0')
    def sg0(self, subgoals):
        if int(self.goal.params['types']) & 1:
            self.satisfy('VideoSearch',**self.goal.params)

    @subgoal('sg1')
    def sg1(self, subgoals):
        if int(self.goal.params['types']) & 2:
            self.satisfy('ImageSearch',**self.goal.params)

    @subgoal('sg2')
    def sg2(self, subgoals):
        if int(self.goal.params['types']) & 4:
            self.satisfy('NewsSearch',**self.goal.params)

    @evaluation
    def ev1(self,subgoals):
        videos,images,news = [],[],[]
        ...
        root = doc.documentElement
        for i in range(0,9):
            if i < len(videos):
                root.appendChild(videos[i])
            if i < len(images):
                root.appendChild(images[i])
            if i < len(news):
                root.appendChild(news[i])
        self.props.results = root.toxml('utf-8')
        ...

    def commit(self, subgoals):
        self.timescommitted += 1

```

Figure 2-13: MixVidImgNews Technique Module

Python decorators are syntactic sugar to take the subsequent function definition as input and return a wrapped version. The Technique class provides a *get_eval_stages* method for the Planner to obtain and execute each stage.

The MixVidImgNews Technique shown in figure 2-13 uses three subgoal decorators to execute each of the searches for videos, images and news. Each decorator takes in an object that contains subgoal properties keyed by the subgoal name. After a subgoal is executed, its properties are accessible to the current and subsequent stages. The *ev1* stage accesses each subgoal's search results and returns XML mixed with videos, images, and news articles.

The Planner recursively iterates through all Goals, Techniques and subgoals during execution using the *iterate* method. The top-level Goal is received as a query parameter within the AJAX request, and the other query parameters are passed to the Goal as inputs. The TechniqueNode and GoalNode classes wrap Techniques and Goals. Subgoals are satisfied by a *satisfy* method set dynamically on the TechniqueNode at runtime. The *satisfy* method is created using Python's *partial* method from the *functools* module. The *partial* method returns a method with the subgoal parameter name already bound.

To complete Goal execution, the Planner recursively invokes *commit* methods on Goals and Techniques. Commit methods return snapshot trees that reflect the final state of execution. A GoalSnapshot contains the underlying Goal object, a snapshot of the winning Technique, and a list of snapshots for the other Techniques. A TechniqueSnapshot contains the underlying Technique object, a dictionary of subgoal snapshots, and a view of its properties. After committing the plan we obtain a Goal tree represented by the top-level Goal snapshot. We recurse through the entire Goal tree and convert it into a dictionary. A Python print statement called on a string representation of the dictionary sends the JSON Goal tree to the client. Figure 2-14 shows a fragment of the *rungoal* code.

```

setg = "g = "+chosenGoal+"("+repr(params)+)"
exec(setg)
plan = planner.satisfy(g)
plan.iterate()
topgoalsnapshot = plan.commit()
goaltree = createGoalJSONelt(topgoalsnapshot)
print(str(goaltree))

```

Figure 2-14: Run Goal Code

2.8 Goal Trees

Goal trees allow users to conceptualize how the Planner behaves on search requests. Each Goal's winning Technique is listed first and the other Techniques listed afterwards. Figure 2-15 shows the Goal tree for Mixed Search Goal.

A *Show goal tree* check box appears on the home page and causes the browser to open a new window that displays the Goal tree. Javascript's `window.open` opens a new window and HTML is written directly to the window's DOM document. We populate the HTML head tag with our CSS style sheet, two script tags and a reload script tag. The first two script tags dynamically load the jQuery and treeView Javascript libraries and the reload script reloads the page to properly display jQuery's treeView in Internet Explorer. We offload tree views to new windows to avoid loading jQuery libraries on our home page and to give users more space to view attributes.

jQuery treeView allows users to expand and collapse tree elements with mouse clicks. We interface with jQuery treeView by creating a bulleted list using HTML `ul/li` tags. We create methods in Javascript that take in elements of the JSON Goal tree and output formatted bullets. The root of the JSON Goal tree is passed to *creategoalli*, which formats the Goal name and calls *createparamsli* and *createtechli* to format its parameters and Techniques. The *createtechli* method formats its attributes with *createattli* and calls *creategoalli* for its subgoals. Figure 2-16 shows a screen shot of jQuery treeView. Goal names are gold and Technique names maroon.

We make Technique code viewable through the browser. When a user clicks on a Technique in the Goal tree, a new window opens with the code. The viewable code has search examples and allows users to share code. We preserve backslashes, quotations and newlines by escaping and use HTML `code` and `pre` tags to display code in



Figure 2-15: Mixed Search Goal Tree

the browser.

2.9 Heuristics and Biasing

Goals provide a heuristic formula to evaluate Technique attributes. The evaluation formula is in the Goal's module. Evaluation code can be changed without change to the server. Video Search Goal uses the *toprating* attribute to evaluate search results. The rating is obtained from the highest ranking video. YouTube returns a user rating for each video for how online users rate the video. Truveo returns a proprietary *vRank* rating, and user ratings are returned when available from the video's original source. We use a RateResults subgoal to return the max of Truveo's user rating and *vRank*. Google AJAX API returns its own video rating. Ratings are scaled between zero and one and used to determine from which Web service to return results.

Video ratings do not always meet user expectations, and image and news search

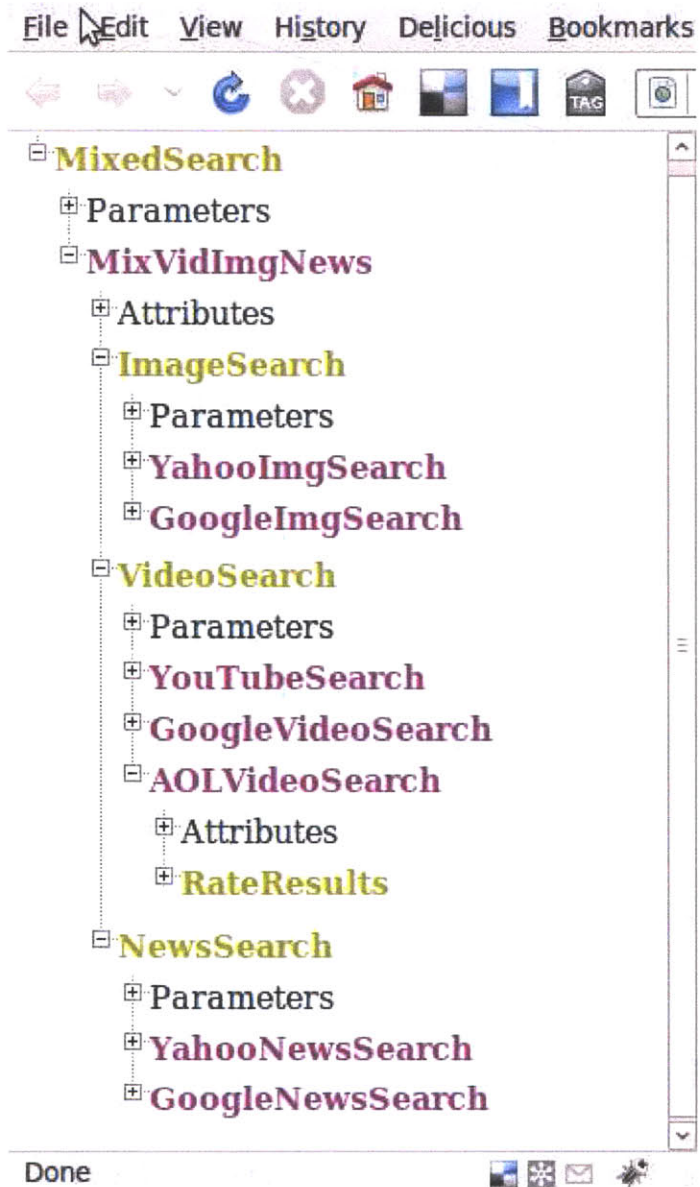


Figure 2-16: Web Goal Tree

results do not return ratings. To address this we allow users to weight Web services. A user may decrease weight on YouTube and increase weight on Truveo so that Truveo search results are more likely returned. We call this biasing. Users may bias toward a single service by weighting competing services to zero. Users have complete control over which search results to display. Biases are keyed by Web service URL and are sent to the server as Goal parameters.

We display bias sliders for the user to drag with the mouse. The slider is implemented by attaching Javascript functions to the DOM document's *onmousedown*, *onmouseup* and *onmousemove* event handlers. The *runnable* CSS class changes the cursor to *move* on mouse hover to alert the user to drag. We check that the event's *target* is a slider, save the slider's *x* coordinate and set the *onmousemove* handler. The *onmousemove* handler adjusts the slider's position each time the move event fires. Finally, we remove the *onmousemove* handler and set the corresponding bias input to the slider's final position. In browsers that support it we set the mouse type to *-moz-grabbing* while the user drags.

The HTML layout includes only those Web service biases relevant to a particular search. We do not display Web services that search images when the user chooses to search videos or news. To do this we key every Web service's *span* element *id* with the URL concatenated to the Goal name. When the user chooses search content, the Web service's CSS *display* property is switched between *inline* and *none*. We display the Web service name above each slider and display the source URL as a tooltip on mouse hover. Web service names are user friendly and URLs serve as unique identifiers.

2.10 Adding Techniques

The Planner is extensible and we provide mechanisms for users to add their own search implementations. Users can view returned attributes within jQuery's *treeView* to see what attributes and formats are required. For example, the Video Search Goal has two required attributes. The *videos* attribute is an XML formatted string of the search results and *toprating* is used for Goal-level evaluation. The XML string is passed to

the HTML display code and parsed using Python's *xml.dom.minidom* module.

2.10.1 Uploading

Users upload Techniques by entering an additional repository on the home page. The repository is saved in a client cookie and may be changed. The server checks for the cookie and pulls in the new Techniques. For our demo we have an additional repository that contains the Technique to search videos using the Google AJAX API. Chapter 3 discusses security concerns and remedies for uploading and executing untrusted code.

2.10.2 Debugging

Users may debug their Techniques using the jQuery Goal tree. The Planner was modified to run each Technique stage within a try/catch block. If an exception is thrown a special *exception* attribute is added to the Technique's Goal tree. The *exception* attribute is populated with the exception type, exception message, Technique stage and line of code. The line of code is obtained from the traceback returned by `sys.exc_info()`.

2.11 Saving Settings

We use cookies to save user input. The *Remember values* checkbox saves user biases and the query string. The values stored in cookies are saved for 31 days and are pre-populated when the user reloads the home page. Users may save settings to remember their biases, to begin where they left off or to save their favorite searches.

2.11.1 Cookies

We have Javascript utility methods to get, set and delete cookies. The utilities are in a separate file and are accessed by the main page and jQuery treeView. Javascript cookies are available in *document.cookie* as a semi-colon separated list of name/value

pairs. Our *setCookie* method sets the expiration using the Javascript Date object by adding the equivalent number of milliseconds to the current time. Cookie values are escaped using the *escape* function so that special characters are replaced with their *%hex* equivalent for network transfer. The cookie value is unescaped in *getCookie*. To delete a cookie we set its expiration to a time past. The *document.cookie* is parsed as, displays as and is a *typeof* string, but it is uniquely implemented in Javascript to prohibit direct assignment of arbitrary values.

On the server we access the HTTP cookie by checking the “HTTP_COOKIE” environment variable within the *environ* attribute on the *os* module. The *os* module is for miscellaneous operating system interfaces. We create a SimpleCookie using Python’s Cookie module and access cookies by name using Python’s *//* indexing operator.

2.12 HTML Display

We dynamically generate HTML to display search results. Developers specify a default site in Goal specifications or they enforce Techniques to set the *html* attribute. The MixVidImgNews Technique for Mixed Search sets the *html* attribute when videos and images are searched together. The top-level Goal’s results are displayed and sub-goal HTML pages are ignored. Parameters are passed in an HTTP POST request via an HTML form. We use Python, but sites may generate HTML using any programming language that supports HTTP.

2.12.1 Dynamic Layout

The HTML layout is robust and does not require code modification when new Techniques are added at runtime. We iterate over Techniques and adjust the layout accordingly. Both the home page and results page dynamically render their layouts.

2.12.2 Home Page

The home page displays an image of a planner to signify that no search content is chosen. As the user chooses search content, the image is changed to reflect the corresponding search Goal. Web services and biases are dynamically added or removed based on the user's search content. The user repository input moves up or down based on the size of the Goal box. HTML links, parameters, attributes, Techniques, display names and Web services are indexed by Goal name in Javascript JSON objects. We store the name of the current Goal in a global variable and use Javascript's *eval* function to retrieve Goal data. Goal boxes are headed by their display name and list Web service biases and Goal parameters. The Goal's attributes are sent to the HTML page using an HTML form. Figure 2-17 shows a screenshot of our home page. The News Search Goal box reflects that news content is checked. The Mixed Search Goal box is displayed when two or more types of content is searched.

2.12.3 Results Page

We use separate pages to display videos, news, images and mixed search results. The Mixed Search Goal page displays combinations of content. Pages for the other Goals display one type of content. Page layout is finely formatted to the type of content and special cases are avoided. Search results are tabbed to show pages of results. Tabs are HTML anchors that switch the CSS *display* property between *block* and *none*. Each tab is accessed via a numbered link at the top of the page. The CSS *display* property set to *none* pulls an element out of its normal flow so that it takes up no space on the page. Videos and images display nine results per page. News results display six results per page to handle larger titles and to show portions of the article description. We use Python's *xml.dom.minidom* to parse search results. Figure 2-18 shows a screenshot of tab 1 for video search query "Mit Dome".

Planner

Goal Driven Web Search

search news

Sources

Yahoo! News Bias 0.44

Google News Bias 0.8

text

Show goal tree Remember values

Choose content to search:

Videos
 Images
 News

Your Repository:

Figure 2-17: Home Page

2.12.4 Cascading Style Sheets

We store a separate file for CSS styles in a directory under the site's root directory. Using a separate file is good programming practice to cleanly separate logic from display. Styles in external style sheets are shared by all files. We define CSS classes on various HTML tags that are set on HTML elements. We use *div* elements to layout our page instead of tables because *divs* allow us to finely position elements with absolute and relative positioning. We center *div* elements by setting the *margin-right* and *margin-left* attributes to *auto* as specified by W3C. Absolute positioning is used to move elements anywhere on the page. Relative positioning is preferred for small tweaks when moving elements relative to their original position. We handle differences in how Firefox and Internet Explorer render positioned elements.

Video Search

1 2



0:47
[MIT Red Line](#)
2213 views



2:20
[Turning MIT's](#)
416 views



0:43
[MIT Red Line](#)
1973 views



1:51
[Queensberry](#)
1649 views



3:48
[Curse mit Si](#)
2050 views



4:31
[The Dome 50](#)
5110 views



9:55
[Gterzge Rhei](#)
13825 views



3:16
[The Dome 51](#)
7023 views



1:48
[MIT Great Do](#)
432 views

Figure 2-18: Video Search Results

Chapter 3

Security Issues

We defend against Cross-site Scripting (XSS) and Cross-site Request Forgery (CSRF). Quotes are escaped and HTML tags and attributes are removed from user-supplied inputs. This prevents attackers from running Javascript code to corrupt cookies or site layout. Attackers are prevented from triggering the *rungoal* script. An attacker could execute the Mixed Goal search to implement a DoS against our site. We prevent CSRF by checking that the HTTP referer header refers to our site.

We upload third party code to execute on our server, which makes our system vulnerable to malicious code. Python is a flexible language because it is not statically typed, has no coherent notion of private variables and shares sensitive modules such as *sys* between all modules. We do not service requests as root and allow minimal access to the file system. Malicious code may tamper with the file system, debug or kill running processes, corrupt kernel memory and builtin Python modules, or simply hang indefinitely. We present a threat model and discuss possible remedies to the problem of running untrusted code.

3.1 Threat Model

Our Web application runs as the *www* user on the Apache server. The *www* user has fine-grained access to certain directories on the file system and complete network access. It is foreseeable that an attacker may achieve root access, in which case he

has access to the entire file system and all system calls and messaging facilities, for example to kill or debug other processes. Code running as root could be sand-boxed in Python. We focus on viable threats when untrusted code is run as non-root. The following are presumed about an attacker

- An attacker may loop indefinitely
- An attacker may import our module and clobber attributes that point to sensitive functions and data
- An attacker may clobber shared Python resources required for proper functioning of the application such as *sys* module attributes
- An attacker may access files accessible to the *www* user
- An attacker may make arbitrary network calls to download external code, perform DoS, send email, etc.
- An attacker may freely use Python's introspection capabilities to access objects, variables and code anywhere in the interpreter
- An attacker may use Python's *eval* and *exec* functions to execute arbitrary Python code

3.2 Running Untrusted Code

3.2.1 Sandboxing

Sandboxing is a broad term that applies to code run in a restricted environment with limited access to privileged operations such as file system and network access. Some well known sandboxes are Java applets, Google Native Client and virtual machines. One sandboxing model that runs Python is Google App Engine. Google App Engine allows developers to develop Web applications that run on Google's infrastructure. Developers download an Integrated Development Environment written in Python to

develop their application and upload their application to Google servers. Here is a list of sandboxing techniques that Google App Engine implements:

- Restricts access to the file system
- Confines outside communication to standard URL fetch and email services
- Designates a single data store to persist data
- Disallows process spawning
- Runs applications only in response to Web requests
- Kills applications that delay returning a response
- Confines incoming connections to HTTP and HTTPS
- Disables unsafe Python imports

This approach runs untrusted code in its own process. There are cases where we'd rather sandbox code within an existing Python interpreter. This more fine-grained sandboxing is helpful for the Planner because it allows us to import untrusted code and directly call its methods using standard Planner interfaces. Closures can be used to enforce access control at the application level. Tokens is a way to implement access control at the interpreter level. The downside is that access control does not by itself protect against malicious code that loops forever or over-consumes resources. However, they do allow us to restrict functionality and access to privileged operations.

Python has access control built into its interpreter. Setting the global `__builtins__` variable to `{}` enables Python Interpreter's Restricted Execution (PIRE) [9] which enforces access control on builtin object attributes. This can be used as basis to develop a capability-based access control system within Python [9].

3.2.2 Closures

Closures have previously been explored in Python as a tool to wrap privileged operations before passing them to untrusted code [3]. A closure is a block of code that can

```

def f(f_open, base):
    def restricted_open(path):
        f_open(base + path)
    return restricted_open

ropen = f(open, '/jaildir')
ropen.func_closure[0].cell_contents <- Introspection

```

Figure 3-1: Python Closures

be transferred. In python it is implemented by creating a nested method definition as shown in figure 3-1. With the help of PIRE [9], the untrusted code can be given a capability, i.e. a closure function, that restricts functionality. In this *restricted_open* example, we prefix all file system accesses with a base directory.

Python allows developers to introspect modules and objects in interpreter-level and application-level code. Without enabling PIRE, untrusted code can obtain closure internals through the *func_closure.cell_contents* attribute. Furthermore, it is difficult to ensure that all references to *open* are inaccessible. For example, every module has a reference to *__builtins__*, so after importing the *urllib* module I can reference *open* via *urllib.__builtins__[‘open’]*. Python objects recursively reference one another to unspecified depths and a diligent search may uncover references to functions that were hidden at higher levels. Use of tokens is one approach to defend against Python’s extensive introspection.

3.2.3 Tokens

Woodrow, et. al. [11] proposed tokens to enforce access checking at memory level to avoid accidentally exposing sensitive references. Tokens represent access permissions for a module and are assigned to every object at object creation. By default untrusted code may only access objects it creates. The token is checked at the byte-code layer in opcodes that perform operations on memory such as *LOAD_GLOBAL*, *STORE_GLOBAL*, etc. The tokens were prototyped on PyPy but can be ported to CPython. We can use tokens to enforce that all sensitive objects remain inaccessible to untrusted code. This comes at a performance cost because an extra check has to be made on all memory accesses. Table 3.1 shows which opcodes must perform access

checks or token updates to prevent access breaches by untrusted code.

3.2.4 Other Python Solutions

Python has application-level modules to enforce access control and to restrict functionality to untrusted code. The Zope3 [12] package runs untrusted code in a separate interpreter. Objects that provide sensitive functionality are wrapped by the application to mediate access to methods and attributes. The untrusted code sets variables that are accessible to trusted code after code execution. Zope3 prohibits *exec* calls and use of try/catch blocks. RestrictedPython [8] compiles untrusted code in an environment with user supplied implementations of print, import, getattr, setattr, etc. and then executes it using Python's *exec* method. No convenient hooks are provided for the application to interact directly with the untrusted code. Python *cyptes* can be removed from the interpreter to make certain types private or otherwise inaccessible to untrusted code. PyPy [6] is an ongoing project to provide a framework for building dynamic programming languages. PyPy is a Python interpreter written in the Python programming language. PyPy's sandboxing features allow the programmer to compile untrusted code in a separate interpreter. The programmer communicates with the interpreter using stdout and stdin [7]. A controller examines and sanitizes what the untrusted code sends to the trusted code.

Opcode	Operation
BINARY_SUBSCR	Access check
STORE_SUBSCR	Token update
DELETE_SUBSCR	Access check
STORE_NAME	Token update
DELETE_NAME	Access check
UNPACK_SEQUENCE	Access check
FOR_ITER	Access check
STORE_ATTR	Token update
DELETE_ATTR	Access check
STORE_GLOBAL	Token update
DELETE_GLOBAL	Access check
LOAD_NAME	Access check
LOAD_ATTR	Access check
LOAD_GLOBAL	Access check
LOAD_CLOSURE	Access check
LOAD_DEREF	Access check
STORE_DEREF	Token update
MAKE_FUNCTION	Token update
MAKE_CLOSURE	Token update

Table 3.1: Python Opcode Token Accesses

Chapter 4

Alternative Designs

We experimented with returning XML responses for our AJAX Goal invocations. The time to return and process XML responses proved intractable. To create and parse the entire XML Goal tree degraded performance despite the presence of Javascript and Python functions to parse XML into Document Object Model (DOM) objects. Python and Javascript support JSON innately as builtin objects which allows for fast creation and evaluation of JSON strings. JSON outperformed XML ten-fold. We did not keep measurements of XML latencies, but the browser stalled for over a quarter minute in some cases.

We initially put all Goal boxes on our layout sequentially in rows of two on the home page. It became clear that we needed a simpler layout to consolidate the many buttons, sliders and text boxes spread throughout the page. As more Goals are added to the application, it is overbearing to have so many titles and inputs showing simultaneously. To solve this problem we use the CSS *display* attribute to switch between the Goal boxes of different Goals.

We considered SOAP and XML-RPC as a way to make the Web application extensible. Each Technique would communicate with the Planner over SOAP or XML-RPC and third party implementations would return XML for platform and language independence. We'd have to build an entirely new abstraction layer and protocol on top of the Planner to interface with any number of Goals and Techniques. We gain in portability but lose in performance and development effort. The overhead

was determined to have minimal advantage for our simple application especially when Python and JSON are simple alternatives.

We considered using paging available through the YouTube API to page through results. This would allow users to page through dozens of results instead of just 15. We chose not to implement paging for compatibility purposes because not all Web services support paging. We also identified sluggish paging in other sites where users wait for the next page to load. To remedy slow load times, we can pre-fetch the next page before the user requests it, and we are always one step ahead of the user. This would help avoid degrading the user experience.

We looked into Web templates in Python Web frameworks such as Django, Mako, Jinja, Kid and Cheetah. These frameworks allow application code to be interleaved with HTML layout and provide other useful Web development tools. Python supports UI/logic interleaving innately via print statements. This fact, and the desire to retain fine control over our call stack convinced us to forgo using Web frameworks. The pros and cons of various Web frameworks can be researched in future work.

Chapter 5

Future Work

Future work includes research to dynamically add Goals, port scripts to a constantly running server, and implement Python sandboxing features. It became clear that to dynamically add Goals is beyond the scope of this project. We make basic assumptions about Goal behavior to mix results and to switch between search content. A video conversion service that converts video formats between Windows Media, flash, MPEG, etc. would require additional logic in both client and server code. We include support for repositories, Goal specifications, and external HTML display pages with the intent that new Goals are added dynamically. The Planner engine is well suited to add new Goals and Techniques dynamically. The challenge going forward is to make the home page independent of search and to seamlessly switch between various unrelated Goals.

The Web application runs on Apache on MIT's CSAIL infrastructure and invokes our code as a script. Scripting alone limits performance and is transient per request. We'd like to have a running server that persists across user requests, uses privilege separation to separate functionality and more directly responds to user requests for content. This will be particularly useful when we add support to add Goals dynamically. We'd like to incorporate access control to preserve the integrity of the application and system when untrusted code is executed. Preferably we use a capability-based system based on Python closures or tokens, otherwise running untrusted code in a separate interpreter is also an option.

A constantly running server will allow us to more flexibly run Goals and Techniques. For example, we can use the Planner's *update* decorator to gradually refine goals and run them multiple times. Currently we do not use the update stage because we run each Goal "once and done" on each request. We can improve performance by amortizing processing costs across many requests by caching heuristic values or result sets for common requests. This reduces latency when running Goals with multiple subgoals. For example, our MixVidImgNews Technique runs each subgoal to search videos, images and news sequentially, which tallies to searches on seven Web services. Using cached data we can research efficient methods to reduce the number of Techniques and Web services required to process a given Goal request.

We'd like to research ways to more fully sandbox untrusted code running within the same interpreter as trusted code. Access control allows us to make variables private and to restrict functionality, but this does not prevent malicious code from looping for ever. This is an easy way for malicious code to implement a DoS attack against our server. One solution is to filter out infinite loops through static analysis at upload time. A *while(true)* loop that does nothing is obviously malicious and can be removed and we can prevent the module from running altogether for its misdeeds. A second approach is to identify infinite loops at runtime at the bytecode level and forcefully break out of them.

Further research is required to improve heuristics for searching videos, images and news on the Web. We use video ratings, but we do not have relevant heuristics to evaluate image and news searches. We choose the top rating and use it to represent the quality of the entire result set. More intricate heuristics that combine comments, tags, ratings and view counts can be used to test more comprehensive evaluation formulas. We'd like to research ways for users to choose properties used in the evaluation formula.

We want to more fully use the special capabilities of each Web service. For example, YouTube supports paging so that users can page through dozens of results. We want users to view as many results returned from their preferred Web service as possible. The default HTML display for a single Goal handles data in the same way

for each Technique. We want to be cognizant of ways to share code and to combine appealing features from the various Web services.

Chapter 6

Related Work

The Planner [5] was developed at MIT CSAIL to allow applications to dynamically adapt to changing conditions in ubiquitous and pervasive computing environments such as mobile communication. One approach is for applications to declare what they need before execution and the runtime environment determines how to satisfy the declarations. For example, Adjie-Winoto, et. al. describe an Intentional Naming System (INS) [1] in which applications use names to specify intent in the form of resource attributes rather than simply network locations. The names are resolved at runtime at the network layer. Becker, et. al. propose a component-based system for pervasive computing (PCOM)[2] that offers programmers high-level programming abstractions to capture dependencies between pervasive components. The resulting dependency tree supports automatic adaption to changing conditions in the pervasive environment. Costa, et. al. [4] introduce a middleware approach for networked embedded environments that leverages self-configuring sensors and actuators to build a language-independent component-based programming model minimal enough to run on typical network embedded devices.

The Planner has been used for a number of applications. Paluska, et. al. [5] demonstrated the Planner through the JustPlay Audio and Video application to handle automatic configuration of A/V-compatible devices. The JustPlay application uses an RTPStreams Technique for runtime determination of its Real-time Transport Protocol by declaring two subgoals. The first subgoal chooses an RTP AV source

and the second chooses an RTP AV sink. The source and sink subgoals are run in subsequent stages. The chosen source stream format of DarwinStreaming or MythTV determines which of the HDTV or Laptop is chosen as the sink. JustPlay includes a way for third parties to create their own Techniques, particularly, it includes a VLCHost external Technique that dynamically discovers hosts among HDTV, laptops and desktops.

The Planner is also used in other applications to demonstrate its usefulness beyond pervasive computing [5]. The crisis management application allows new strategies to be added as the crisis unfolds. The Recipe application makes recipes from available ingredients to match user-specified preferences. Each recipe is encapsulated in a Technique and available utensils are accessed in subgoals. The Planner has been considered in hardware design. The Planner chooses between various strategies for building circuits and outputs Verilog.

Chapter 7

Conclusion

We have demonstrated that Goal-Oriented search is useful for aggregating and comparing Web services on the Internet. We use the Planner [5] for Goal-driven search to decompose functionality. New Web services are added dynamically without requiring changes to the Web application. Standard Web technologies such as REST, JSON and XML are leveraged so that performance and portability are preserved.

The Web application is decomposed into high-level Goals that are implemented by one or more modules called Techniques. We aggregate Web search results from YouTube, Truveo, Google, and Yahoo. Video searches return ratings that allow us to automate evaluation and comparison of search results. Users bias toward his or her preferred service by adding weights that scale search result ratings.

Repositories declare Goals and Techniques which the Planner uses to execute search queries. Users may add a repository to declare their own search Techniques. HTML display modules are declared separately and other HTML display modules can be declared at runtime. The home page and results pages respond dynamically to modified Techniques and search results.

The application uses AJAX to run Goals and the resulting Goal tree is returned in JSON. Goal trees reflect the entire state of the Goal's execution and are displayed within the browser using jQuery treeView. Goal trees show Goal parameters and attributes, and allow users to debug their own search implementations. Technique code is viewed in the browser for users to examine and add their own.

Bibliography

- [1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. *17th Symposium on Operating Systems Principles*, pages 186–201, December 1999. MIT Laboratory for Computer Science.
- [2] Christian Becker, Marcus Handte, Gregor Schiele, and Kurt Rothermel. PCOM - A Component System for Pervasive Computing. *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications (PERCOM'04)*, pages 69–78, 2004.
- [3] Brett Cannon and Eric Wohlstadt. Controlling Access to Resources Within the Python Interpreter. http://people.cs.ubc.ca/~drifty/papers/python_security.pdf.
- [4] Paolo Costa, Geoff Coulson, Richard Gold, Manish Lad, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, Thirunavukkarasu Sivaharan, Nirmal Weerasinghe, and Stefanos Zachariadis. The Runes Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario. *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications (PERCOM)*, pages 69–78, 2007.
- [5] Justin Mazzola Paluska, Hubert Pham, Umar Saif, Grace Chau, Chris Terman, and Steve Ward. Structured Decomposition of Adaptive Applications. *PerCom*, 2008. MIT Computer Science and Artificial Intelligence Laboratory.
- [6] Pypy documentation. <http://code.google.com/p/google-caja/>.
- [7] Pypy's sandboxing features. <http://codespeak.net/pypy/dist/pypy/doc/sandbox.html>.
- [8] Restrictedpython 3.5.1. <http://pypi.python.org/pypi/RestrictedPython/>.
- [9] Tav. Paving the Way to Securing the Python Interpreter. AskTav!, February 2009. <http://tav.espians.com/paving-the-way-to-securing-the-python-interpreter.html>.
- [10] Xmlhttprequest W3C Working draft 19 november 2009. <http://www.w3.org/TR/XMLHttpRequest/>.
- [11] Stephen Woodrow and Victor Williamson. Securely Executing Untrusted Code in Python. MIT 6.893 Systems Security Final Project, December 2009.

[12] Zope 3 API documentation. <http://docs.zope.org/zope3/>.