Global Information Assurance Certification Paper

Obfuscation and Polymorphism in Interpreted Code

GIAC (GSEC) Gold Certification

Author: Kristopher L Russo, xythex@live.com
Advisor: Dr. Johannes Ullrich
Accepted: January 11th 2017

Abstract

Malware research has operated primarily in a reactive state to date but will need to become more proactive to bring malware time to detection rates down to acceptable levels. Challenging researchers to begin creating their own code that defeats traditional malware detection will help bring about this change. This paper demonstrates a sample code framework that is easily and dynamically expanded on. It shows that it is possible for malware researchers to proactively mock up new threats and analyze them to test and improve malware mitigation systems. The code sample documented within demonstrates that modern malware mitigation systems are not robust enough to prevent even the most basic of threats. A significant amount of difficult to detect malware that is in circulation today is evidence of this deficiency. This paper is designed to demonstrate how malware researchers can approach this problem in a way that partners researchers with vendors in a way that follows code development from ideation through design to implementation and ultimately on to identification and mitigation.

1. Introduction

Malware infection vectors have shifted over the last few years in response to better detection and remediation capabilities being implemented by heavily targeted industries. These industries include healthcare, retail, government sectors and many others. Malware prevention continues to be the primary focus in the field of information security, and many effective malware mitigation tools have been developed to prevent infection of target devices by compiled code. Most of these tools utilize signature based detection engines (Sophos, 2013) Although a few vendors are taking different approaches to the problem. A notable example is the Next-Generation Endpoint Security space where companies such as Carbon Black have focused on enhancing signature-based detection by using cloud sourced and reputation based services to quickly identify new threats (What We Do, n.d.). Other vendors have tried to reduce time to detection and the risk presented by polymorphic threats by searching for the signatures of malicious features and functions instead of evaluating the entire code sample (Predictive Malware Detection). Some vendors are moving away from signatures entirely by detonating unknown binaries in a virtual environment and then using artificial intelligence and machine learning to evaluate the resulting behavior to look for malicious intent. Cylance believes that this approach can decrease complexity and increase remediation effectiveness significantly (Cylance).

In response to these advances in prevention, malware writers are increasingly turning to macro infection techniques and other interpreted code approaches. Exploit kits are used to generate zero-day code by relatively unsophisticated threat actors. Interpreted zero-day code is subsequently embedded in documents and then distributed via channels trusted by most users; channels such as e-mail, advertising networks, and content management portals (Mell, Kent, & Nusbaum, 2005).

One of the most popular of these code interpretation engines, the Windows Scripting Host, is ubiquitous on modern Windows-based computers. The Windows Scripting Host does not contain any native authentication ability and allows a broad

access to the underlying Operating system via the Windows Management Instrumentation interface and COM libraries.

However, interpreted code is not immune to signature based detection. Instead of looking at a code sample in its entirety anti-malware can look for the signatures of potentially malicious functions. To bypass this detection approach, malware authors are increasingly turning to code obfuscation and in some cases the polymorphism of code. Since this requires knowledge of both coding techniques and the underlying operating system it has been difficult, time-consuming, and still somewhat uncommon. It is, however, very effective and is quickly gaining popularity.

With the rapid evolution of exploit kits, skilled coders can create black-box frameworks that handle the complexity of malware generation and then either distribute the framework or offer access to it as a service. These frameworks expose a very broad audience of threat actors to sophisticated infection techniques that would otherwise be beyond their skill level. By using these frameworks, threat actors can easily generate zero-day exploits that can bypass all but the most sophisticated of anti-malware engines.

This paper demonstrates a framework built to create obfuscated polymorphic code that uses the Windows Scripting Host to deliver a payload. This framework is simple in its design and modular so as to allow easy modification and expansion. The easy modification of the core framework is a feature recently seen in the wild that allows old threats to evolve quickly in response to advances in anti-malware. Traditional antivirus products and many next generation solutions do not detect the code generated by this framework. The fifty-six anti-malware products featured on virustotal.com at the time of this writing did not flag the code generated by this framework. Note that the framework in this paper has been purposely crippled and will only generate code that can morph a maximum of two iterations.

The goal of this paper is to help convince industry leaders and researchers that open research, communication, and cooperation are essential in the battle against malware. No longer can solutions and research be kept confidential and hidden. Effectively combating modern threats will require that defenders think and work much more like the enemy by collaborating and expanding on each other's ideas to create

solutions that are quicker to market, less brittle and more robust than the tools available today.

1. Code Framework

1.1. Obfuscation

While there are many different approaches to obfuscating interpreted code, several stand out. Base64 encoding is perhaps the most popular method of obfuscation but is easily identified by even basic anti-malware engines. Most anti-malware products recognize the block size of Base64 code and its telltale signs of padding. This obfuscation method gets used so much that it is likely to trigger intrusion detection systems and anti-malware engines that might otherwise miss the test code (Fiscus, K., 2011, April 13). Therefore this approach was not considered for this project.

Another popular way of obfuscating VBScript is the Windows Scripting host's built-in encoding engine. Again, this draws unwanted attention to the code and provides little in the way of protection (indogeek, 2015, October 15). Another common method of obfuscating code is character substitution. Character substitution can be a very effective technique due to the resource intensive process of scanning and then recombining character codes into readable text. Specifically, this is very effective at hampering signature based detection platforms by intermingling encoded, and plain text without impacting the functionality of the code. Code padding can overburden scanning engines to the point of impacting system performance. Anti-malware engines must make a tradeoff between security and system performance which can be exploited by malware. The problem with this approach is that some anti-malware platforms automatically treat character mapping functions as a likely indicator of malicious code (Ginos, A. 2010 April 1). This approach became key to the framework presented in this paper.

To effectively evade signature based anti-malware engines, this project required the creation of a custom encoder/decoder. This framework utilizes a symmetric system to avoid the complexity of key management. Asymmetric key management routines can be a dead giveaway that encryption routines are present in the code and they add unnecessary bulk to the framework and generated code. The Windows Scripting Host has

native support for the encoding scheme featured in this framework through the chr() and asc() functions. One solution to evading heuristic based detection engines used in this framework is to separate function from content. The strbObfus function of the code sample contains the encoding engine:

```
strbObfus="For Each line In arrInp:For i = 1 To Len(line):workingchar =
asc(Mid(line,i,1)):a = a & Len(workingchar) & workingchar:Next:a = a & 210:Next"
```

This encoding function takes the decimal character designation of each ASCII character in the source script and prepends it with an identifier. The entire chain is then pre-pended to the obfuscated file. Using this approach allows control and non-interpreted characters to be sprinkled into the code to create a large degree of entropy in the finished product with no change to the source's functionality. Many anti-malware and intrusion detection engines simply cannot handle variables of this size and are likely to cease scanning the code without ever scanning the decoding function at the end of the variable. This approach helps to defeat malicious function based scanning engines.

Function brevity is key to avoid creating unique and identifiable signatures. Malware detection engines can trigger on these signatures in obfuscated code. Therefore each function is contained within a single line. The framework is highly variable so that its functions can easily be modified. Additionally, only the decoding engine is included in the obfuscated script stage which prevents the much longer encoding engine from being exposed:

```
strDeOb="b=1:Do:c=Mid(a,b+1,Mid(a,b,1)):b=b+1+Len(c):d=d&Chr(c):Loop Until b>Len(a)"
```

The decoding engine is extremely lightweight but flexible while still being restricted to a single line. It only utilizes the native functionality of the Windows Scripting Host to avoid being flagged as a signature by malware defense systems. The decoding function can parse the encoded script very quickly and then execute the embedded action within the source script. That said, it is still the system's weakest link and the most likely to be detected.

For the code to maintain a minimal footprint on the target system, it had to depend on the Execute function in VBScript. The Execute function evaluates strings as executable at runtime. Even heavily obfuscated strings still parse properly.

Unfortunately, there are limitations to the Execute function such as extremely limited scoping options. Additionally, executable strings cannot store custom functions and subroutines. It is also very difficult to escape double quotes within the string and have it still evaluated as a string when desired and executable code at other times. The polymorphism section covers these issues in depth.

1.2. Polymorphism

The problem with the polymorphic code is that it needs to maintain its base structure to complete the morphing process, but this base structure can be detected and then keyed off of by signature-based detection engines. Additionally, the function's results can trigger behavior based analytic engines. The only perfect form of code polymorphism is the quine which overcomes these issues.

Quines are programs that can reproduce themselves entirely (biv, r. g., 2013 October). There is very little documentation on how quines are coded, and no working examples of VBScript or other interpreted code based quines. Quines themselves are a paradox because no code should be able to understand itself fully (iiSir, ., Foonly, ., Ritemate, S., et el, 2008, April 18). Paradox aside, quines are quite adept at polymorphism, able to not only reproduce themselves exactly but to introduce entropy in each iteration without suffering source degradation.

Once again, the Execute statement in VBScript turned out to be key to solving the quine paradox. By declaring each function as a string, it became possible for the code to reflect on itself as it executed. There were some weird quirks with this approach, the most frustrating of which was a code rot phenomenon occurring with double quotes. As each string containing double quotes was parsed it would lose a set of double quotes. After a few iterations, the code would rot to the point of being non-functional. This behavior was not evident in the logical pseudocode and thus is purely an issue of the Windows Scripting Host.

The solution to the code rot problem was a simple substitution routine. This is demonstrated in the strWrt and strArrInp functions of the code. Here, using the pipe character (which is unlikely to come up in normal VBScript) quotes can be symbolically

represented and replaced at runtime using the VBScript replace() function without them being interpreted by the Windows Scripting Host:

```
strWrt="Set ObjFSO = CreateObject(|Scripting.FileSystemObject|):Set objFle =
objFSo.CreateTextFile(|c:\temp\quine.vbs|,True):objFle.Write
|a=|||&a&|||:|&strDeOb&|:Execute d|"

strArrInp="arrInp=split(|strDeOb=|||&strDeOb&||||&vbCrLf
&|strbObfus=|||&strbObfus&||||&vbcrlf&|strWrt=|||&strWrt&||||&vbcrlf&|strArrInp=|||&strArr
Inp&||||&vbcrlf&|Execute Replace(strArrInp,|||&Chr(124)&|||,Chr(34))|&vbcrlf&|Execute
strbObfus|&vbcrlf&|Execute Replace(strWrt,|||&Chr(124)&|||,Chr(34))|,vbCrLf)"
        Execute Replace(strArrInp,"|",Chr(34))
        Execute strbObfus
        Execute Replace(strWrt,"|",Chr(34))
```

2. Proof of Concept

Once the obfuscation and polymorphism functions were written the rest of the coding went relatively smoothly. A couple of simple ad-hoc debugging functions had to be written and introduced to study the code while running to test the obfuscation features at various test points. However, due to the minimal size and relative simplicity of the framework debugging and functionality testing required minimal effort.

After testing, the code was purposely crippled to prevent its exploitation in the wild. The flaw ensures code replication is limited to two iterations before failure. Uploading the code to virustotal.com in its non-obfuscated as well as its obfuscated form after one morph produced the below results.

Figure 1 shows that the non-obfuscated code being flagged by one heuristics based engine as malicious, the specific reason is not cited. However, a single flag is unlikely to trigger most malware defense products, and the code would end up running in most environments.

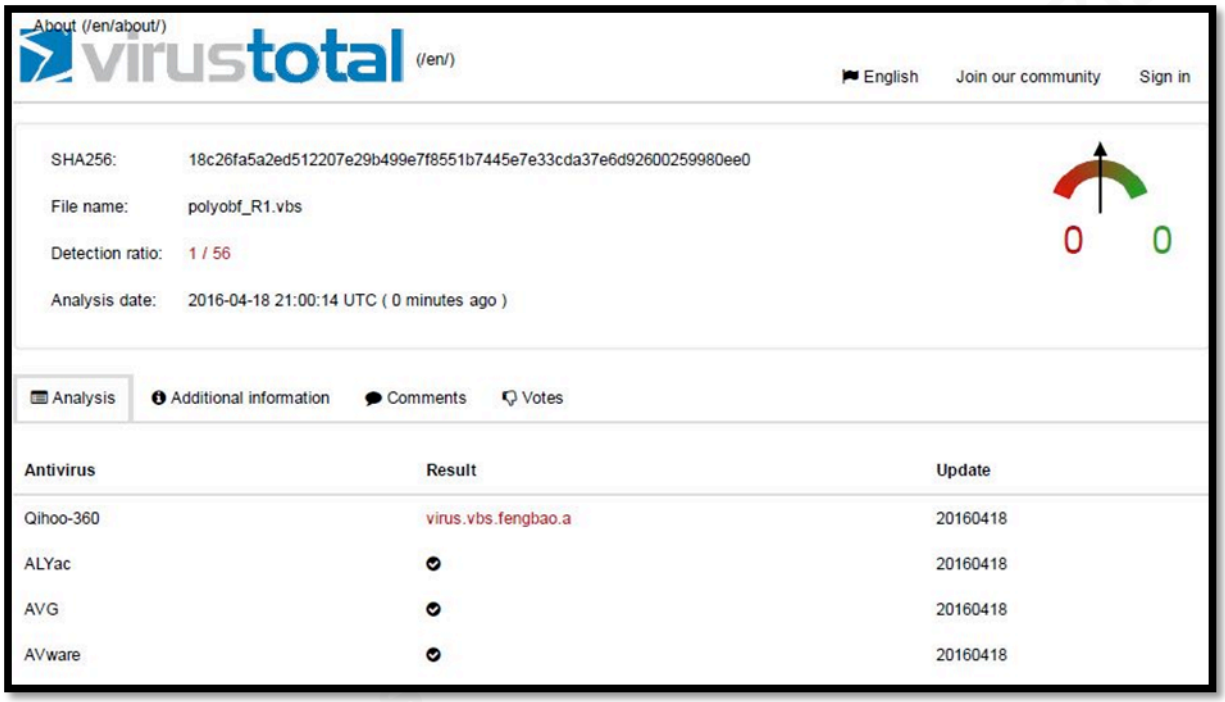

Figure 1. – Virustotal flags the non-obfuscated first iteration of code as malicious by one heuristics engine

Figure 2 shows no detection of the obfuscated and morphed code which demonstrates that the obfuscation and polymorphism engines are effective at evading these products.

Figure 2. – Virustotal does not flag the obfuscated second iteration of code as malicious

Kristopher Russo, xythex@live.com

3. Discussion

This framework demonstrates that not only is polymorphism and obfuscation possible in interpreted code, but it can be accomplished entirely by native functionality included in the code interpretation engine. The succinct and compact code written for this demonstration was effectively able to evade all of the anti-malware engines tested.

Basing the code sample on the embedded Windows Scripting Host gives its payload significant functionality on the target system. It can execute in the context of the user, or if combined with a privilege escalation exploit it can run as an elevated user. The use of the Windows Scripting host also allows the code to execute from the browser. Browser based execution may not require interaction from the end user and removes the need for a third-party distribution platform. While most browsers have security measures in place to prevent this type of execution they are often easily bypassed or have already been disabled to enable websites to access legacy functions.

The non-obfuscated code generated by this framework is never written to disk and is therefore not exposed to most anti-malware solutions. The no-execute flag and storing the code withing a single variable assigned to the Windows Scripting host makes identification of the non-obfuscated code unlikely in volatile memory. The approach used in this framework is not the only effective approach to obfuscation. This framework makes heavy use of random characters for padding, but often it can be more effective to pad with legitimate code. For instance, if the code padded itself with the header information of a legitimate file it could masquerade as code that should be running on the system. Obfuscation could also take the form of hollowing out executables on the target system or inserting itself as a function of an existing executable. The framework could also be expanded on to pad both sides of the code with legitimate data. Hiding with data files is the approach used in steganography.

Mixing polymorphism into this project ensures that the framework remains fresh and relevant. Polymorphism will continue to play a role in the future of evolving threats. The design of this framework focuses on fully polymorphic code, but in practice, this is may not be necessary. Partially polymorphic code is often sufficient if it is effective at mimicking legitimate code, at least until anti-malware engines flag the non-polymorphic

part as a signature. Using the polymorphism engine to changes the code to mimic code on the endpoint reduces the likelihood of identification even further. Partial polymorphism would allow for more flexibility and complexity in the framework.

The modular design of this framework allows it to easily be expanded on to add additional functionality with minimal core code alteration. Adding new modules involves just a few lines of code. There is much potential for future research with this project. The payload could be modified to execute additional code on the target system. The de-obfuscation engine could also be further enhanced to prevent its detection.

There is also room for improvement by introducing a mechanism for code distribution. Invoking the Windows Management Instrumentation system could allow this code to self-replicate to other computers on the same network. Combining this with polymorphism allows a unique instance of the code to exist on each endpoint which would make identification difficult with most existing anti-malware tools. The individual instances could be set up to maintain code on each known workstation creating a matrix configuration.

Effective detection of the code generated by this framework would require behavioral analysis on end-points of all processes to try to identify abnormal behavior. This identification system would have to have a previously known good baseline of the target environment. Additionally, it would only work in an environment where endpoints are tightly managed, and application distribution is centrally managed. In most environments, the end-points change too often on a day to day basis to make this a feasible approach. Application Whitelisting might also be an effective mitigation to this type of code, but it would have to do a deep inspection if the target environment uses interpreted code for systems management. The environment would also have to support certificates or some other form of code management to identify legitimate code. In practice, this is very resource intensive and has a high impact on the end user experience and systems administrators. There may also be other, yet undisclosed, mitigations to defend against this type of threat.

Kristopher Russo, xythex@live.com

4. Conclusion

The goal of this project was to serve as a challenge to inspire malware researchers to come up with new and innovative ways of tackling threats. By demonstrating that new exploits are still easily within reach by a few lines of code, it is easy to see that there is still much work to be done. These problems are not solvable by simple iterations of existing technology. It will take new and revolutionary approaches to detecting malicious attacks. These solutions will not be born in a single R&D department but by the combined efforts of countless researchers each doing a little bit and sharing their results. Only by adopting the same collaborative approach that threat actors use to create malicious code in the first place can researchers hope to outsmart them at their own game.

Today's approach of alternating between buying the latest greatest security product and blaming end-users for its inability to protect them also isn't going to solve this problem. Today's malware problems are not budget or user education issues; they are the result of a fundamental breakdown in the entire malware defense strategy.

There is nothing sophisticated or novel in the framework presented in this paper. This framework builds upon years old, well-documented and widely publicized flaws in systems. By making a slight pivot off of work that's freely available a framework can be thrown together that potentially bypasses the security measures put in place by many individuals and corporations. Classic signature detection is no longer good enough for effective defense. Even next generation technologies such as function introspection and application whitelisting are not a magic bullet. To protect against constantly evolving threats

Malware researchers should be encouraged to think outside of the box and challenged to test the limits of available technologies to improve the capabilities of those technologies and prepare them to meet tomorrow's threats. Researchers must band together and freely share and build upon each other's work. They should be encouraged to seek solutions collaboratively and share those solutions to advance the understanding and effectiveness of the entire information security community.

Kristopher Russo, xythex@live.com

References

biv, r. g. (2013, October). Quines.txt. Retrieved February 13, 2016, from http://spth.virii.lu/v4/codes/roy_g_biv/Quines.txt

Cylance. (n.d.). *Prevention vs. Detect & Respond.* Retrieved January 30, 2016, from Cylance: https://cdn2.hubspot.net/hubfs/270968/All_Web_Assets/White_Papers/PreventionvsDetectandRespond.pdf?t=1454116903252

Fiscus, K. (2011, April 13). Base64 Can Get You Pwned. In *SANS Reading Room*. Retrieved from https://www.sans.org/reading-room/whitepapers/detection/base64-pwned-33759

Ginos, A. (2010, April 1). Use offense to inform defense. Find flaws before the bad guys do. In *SANS Penetration Testing*. Retrieved from http://pen-testing.sans.org/resources/papers/gpen/windows-script-host-hack-windows-120189

iiSir, ., Foonly, ., Ritemate, S., swoof, ., chii, ., & zxqart, . (2008, April 18). A vbs to output its own source code. In *whirlpool*. Retrieved February 13, 2016, from http://forums.whirlpool.net.au/archive/959630

indogeek, . (2015, October 15). Polymorphism In JavaScript and VBScript Using Microsoft Script Encoder. In *Indogeeks*. Retrieved from http://indogeeks.com/polymorphism-in-javascript-vbscript-using-microsoft-script-encoder/

Mell, P., Kent, K., & Nusbaum, J. (2005, November). *Guide to Malware Incident.* Retrieved from NIST Special Publication 800-83: http://csrc.nist.gov/publications/nistpubs/800-83/SP800-83.pdf

Predictive Malware Detection. (n.d.). Retrieved January 30, 2016, from Reversing Labs: http://reversinglabs.com/technology/reversinglabs-hash-algorithm.html

Sophos. (2013). *Security Threat Report 2014.* Oxford: Sophos Ltd. Retrieved from https://msisac.cisecurity.org/whitepaper/documents/3.pdf.

What We Do. (n.d.). *Retrieved January 30, 2016*, from Bit9 + Carbon Black: https://www.bit9.com/

Kristopher Russo, xythex@live.com

Appendix

polyobf_R1.vbs

```
strDeOb="b=1:Do:c=Mid(a,b+1,Mid(a,b,1)):b=b+1+Len(c):d=d&Chr(c):Loop Until b>Len(a)"
strbObfus="For Each line In arrInp:For i = 1 To Len(line):workingchar = asc(Mid(line,i,1)):a = a &
Len(workingchar) & workingchar:Next:a = a & 210:Next"
strWrt="Set ObjFSO = CreateObject(|Scripting.FileSystemObject|):Set objFle =
objFSo.CreateTextFile(|c:\temp\quine.vbs|,True):objFle.Write |a=|||&a&|||:|&strDeOb&|:Execute d|"
strArrInp="arrInp=split(|strDeOb=|||&strDeOb&||||&vbCrLf
&|strbObfus=|||&strbObfus&||||&vbcrlf&|strWrt=|||&strWrt&||||&vbcrlf&|strArrInp=|||&strArrInp&||||
&vbcrlf&|Execute Replace(strArrInp,|||&Chr(124)&|||,Chr(34))|&vbcrlf&|Execute
strbObfus|&vbcrlf&|Execute Replace(strWrt,|||&Chr(124)&|||,Chr(34))|,vbCrLf)"
Execute Replace(strArrInp,"|",Chr(34))
Execute strbObfus
Execute Replace(strWrt,"|",Chr(34))
```

polyobf_child_r1.vbs

```
a="3115311631142683101279298261234298261249258268311125829926127731053100240297244298243249244277
3105310024029724429824424924124125829826129824324924327631013110240299241258310026131002382673104311
4240299241258276311131113112232285311031163105310823229826227631013110240297241234210311531163114
2982792983102311731152612342703111311423226929729931042323108310531103101232273311023229731143114
2
7331103112258270311131142323105232261232249232284311123227631013110240310831053110310124125831193
1
1131143107310531103103299310429731142322612322973115299240277310531002403108310531103101244310524
4
2492412412582972322612322972322382322763101311024031193111311431073105311031032993104297311424123
2
2382323119311311431073105311031032993104297311425827831013120311625829723226123229723223823225024
9248258278310131203116234210311531163114287311431162612342833101311623227929831062702832792322612
3
2267311431012973116310127929831063101299311624031242832993114310531123116310531103103246270310510
5310
8310128331213115311631013109279298310631012993116312424125828331013116232311129831062703108310123
2
2612323111298310627028331112462673114310129731163101284310131203116270310531083101240312429925829
2
3116310131093112292311331173105311031012463118298311531242442843114311731012412583111298310627031
0
8310124628731143105311631012323124297261312431243124238297238312431243124258312423831153116311426
8
3101279298238312425826931203101299311731163101232310031242342103115311631142653114311427331103112
2
6123429731143114273311031122613115311231083105311624031243115311631142683101279298261312431243124
2
3831153116311426831012792982383124312431243124238311829826731142763102232238312431153116311429827
9
2983102311731152613124312431242383115311631142982792983102311731152383124312431243124238311829829
9
3114310831022383124311531163114287311431162613124312431242383115311631142873114311623831243124312
4
3124238311829829931143108310223831243115311631142653114311427331103112261312431243124238311531163
1
1426531143114273311031122383124312431243124238311829829931143108310223831242693120310129931173116
3
1012322823101311231082972993101240311531163114265311431142733110311224431243124312423826731043114
2
4024925025224123831243124312424426731043114240251252241241312423831182982993114310831022383124269
3
1203101299311731163101232311531163114298279298310231173115312423831182982993114310831022383124269
3
1203101299311731163101232282310131123108297299310124031153116311428731143116244312431243124238267
3
1043114240249250252241238312431243124244267310431142402512522412413124244311829826731142763102241
2
3421026931203101299311731163101232282310131123108297299310124031153116311426531143114273311031122
4
4234312423424426731043114240251252241241210269312031012993117311631012323115311631142982792983102
3
1173115210269312031012993117311631012322823101311231082972993101240311531163114287311431162442343
1
2423424426731043114240251252241241210":b=1:Do:c=Mid(a,b+1,Mid(a,b,1)):b=b+1+Len(c):d=d&Chr(c):Loop
Until b>Len(a):Execute d
```

Kristopher Russo, xythex@live.com

License

The FreeBSD Copyright

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE FREEBSD PROJECT ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FREEBSD PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the FreeBSD Project.

Kristopher Russo, xythex@live.com