3.6 Eireann Leverett, Reid Wightman/ Vulnerability Inheritance in Programmable Logic Controllers

3.6.1 Eireann Leverett, Reid Wightman

3.6.2 Reid Wightman

twitter: @ReverseICS

3.6.3 Eireann Leverett

Eireann Leverett studied Artificial Intelligence and Software Engineering at Edinburgh University and went on to get his Masters in Advanced Computer Science at Cambridge. He studied under Frank Stajano and Jon Crowcroft in Cambridge's computer security group. In between he worked for GE Energy for 5 years and has just finished a six month engagement with ABB in their corporate research Dept. He now proudly joins IOActive to focus on Smart Grid and SCADA systems.

His MPhil thesis at Cambridge was on the increasing connectivity of industrial systems to the public internet. He focussed on finding the cheapest way to find and visualise these exposures and associated vulnerabilities. He shared the data with ICS-CERT and other CERT teams globally, and presents regularly to academics and government agencies on the security of industrial systems.

More importantly, he is a circus and magic enthusiast, and likes to drink beer.

@blackswanburst

3.6.4 Vulnerability Inheritance in Programmable Logic Controllers

200 Programmable Logic Controller (PLC) models from a variety of vendors rely on the same third party library.This CodeSys Runtime library gives these controllers access to 'ladder logic'. The authors discovered authentication bypass vulnerabilities in this library. An unauthenticated attackercould potentially upload ladder logic to the PLCs or halt the programs presently running. The authors subsequently performed a scan of the complete IPv4 internet (0.0.0.0/0) to identify controllers, potentially providing access to critical infrastructure, and shared that data with trusted incident responders.

- Talk and paper can be downloaded from `http://grehack.org`

Vulnerability Inheritance in Programmable Logic Controllers

Éireann Leverett and Reid Wightman

701 5th Avenue, Suite 6850
Seattle, WA 98104
http://www.ioactive.com

Abstract. 200 Programmable Logic Controller (PLC) models from a variety of vendors rely on the same third party library. This CodeSys Runtime library gives the controllers access to 'ladder logic'[2]. The authors discovered authentication bypass vulnerabilities in this library. An unauthenticated attacker could potentially upload ladder logic to the PLCs or halt the programs presently running. The authors subsequently performed a scan of the complete IPv4 internet (0.0.0.0/0) to identify controllers, potentially providing access to critical infrastructure, and shared that data with trusted incident responders.

Keywords: Programmable Logic Controller, third party dependency, IPv4/0 vulnerability scan, embedded systems, critical national infrastructure, industrial control systems, security, supply chain, vulnerability inheritance, ladder logic, incident response

1 Introduction

This paper is a humble case study in industrial control systems (ICS) security. It descibes the vulnerabilities present in a number of programmable logic controllers (PLCs), and these vulnerabilities were inherited into the PLCs from the third party library.

The paper is comprised of three basic sections:

1. Descriptions of the vulnerabilities found by Reid Wightman.
2. Analysis of a full IPv4 scan for vulnerable PLCs, and some elaboration of working with Incident Response (IR) teams.
3. Discussion of how multiple failures to detect vulnerabliities can occur from development, to supply chain, and finally to open deployment.

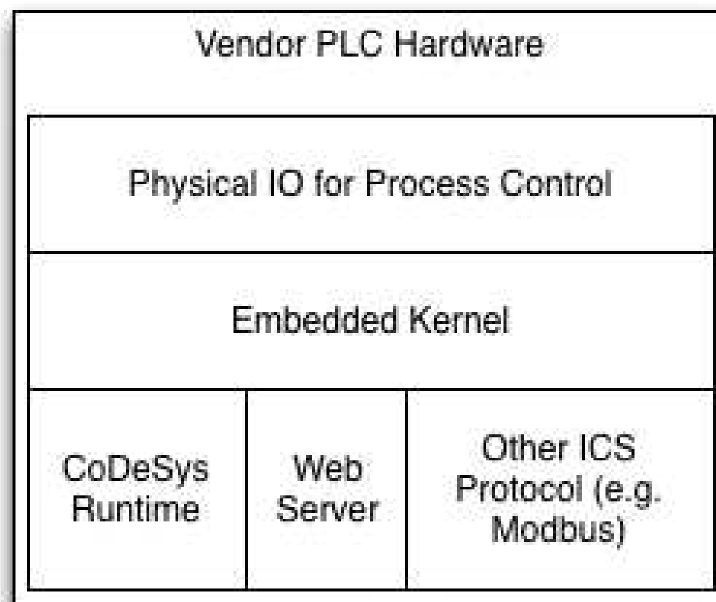
It aims to motivate further discussion of supply chain security and industrial systems security economics.

2 PLCs Inherit Vulnerabilities

2.1 Codesys Software

The CoDeSys PLC Runtime can be found on a wide variety of industrial controllers. Everything from low-end valve control PLCs to substation and syncrophasor management systems use the CoDeSys software. The CoDeSys PLC Runtime also runs on a plethora of hardware – everything from embedded Intel x86 CPUs to PowerPC and m68k CPUs runs the software. There is an entire line of microcontrollers fabricated by a chipmaker that are designed specifically for use as CoDeSys processors[14]. The runtime officially supports embedded Windows CE, Linux, and vxWorks operating systems. 3S Software also produces a 'Soft PLC' which is meant to run CoDeSys logic on desktop and server computer systems. Using Soft PLC, these systems can be used for testing or as a distributed control system.

Fig. 1. CoDeSys as part of a PLCs software architecture



The CoDeSys runtime is meant to lower the cost of development for PLC manufacturers. A manufacturer will typically purchase a commercial operating system such as vxWorks, a separate ICS protocol server such as a Modbus or

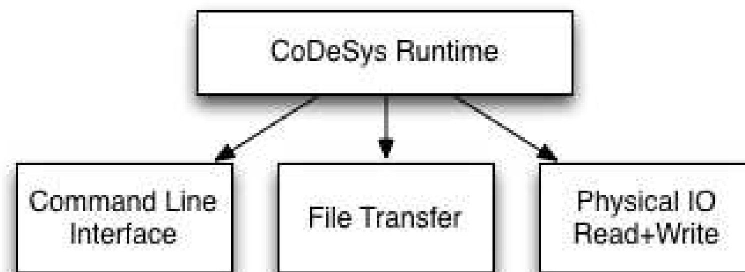
DNP3 stack, and perhaps even use another commercial embedded webserver. Combining these software pieces together allows the manufacturer to focus on developing PLC hardware.

As an added bonus, using the CoDeSys ladder logic runtime means that the PLC manufacturer does not need desktop software for writing ladder logic for their PLC. 3S-Software GmbH, the creators of CoDeSys, produce software for writing ladder logic already. The PC software will compile and load ladder logic into the PLC using a small plugin written by the PLC vendor.

2.2 Vulnerabilities in the design phase

The problem is one of insecurities introduced at the design phase of the authentication protocol.

Fig. 2. CoDeSys services provided by the runtime



The CoDeSys runtime listens on a special TCP port (either 1200 or 2455). This port provides three services for controlling industrial processes. These include a command line interface, a file transfer service, as well as the ability to read and write to the PLCs physical IO.

Capabilities available using the CoDeSys runtime include the ability to stop and start the PLCs ladder logic operation, transferring new ladder logic into the controller, and direct manipulation of sensor and actuator data.

Theoretically, these operations can only be done by an authorized person. In reality, no authentication is required to issue these special instructions[10].

3S Software has produced a patched version of their Soft PLC which enables password protection, but for now embedded products are without mitigation [11]. For these products, the vendor that manufactured the PLC will need to create a new firmware image provide their customers with tools to update the PLC. Firmware updates for PLCs are still an incredibly rare thing, and can be very difficult for end users to roll out.

Another problem with the CoDeSys runtime is 3S Software's method of encoding their ladder logic. Ladder logic is encoded as native CPU instructions for

GreHack

the PLCs microprocessor. Generally the CoDeSys process runs with administrative privileges. This is because the ladder logic must be able to read and write to physical hardware outputs. The easiest way for manufacturers to grant this permission is simply to run the CoDeSys Runtime with Administrator or root privileges. As a result, in addition to uploading new ladder logic, a malicious user could insert a rootkit into the PLC. This rootkit could easily block future logic updates.

Yet another problem with the CoDeSys runtime is the file transfer mechanism. File transfer is used by the PC software to enable the transfer of ladder logic, as well as to read a ladder logic program from the PLC.

Ladder logic is typically installed on a PLC using a file called DEFAULT.PRG. A second file, DEFAULT.CHK, acts as a checksum to ensure that DEFAULT.PRG was transmitted successfully. The file transfer mechanism makes no attempt to inspect for directory traversal. As a result, the CoDeSys runtime will allow reading and writing to any file on a typical PLC.

The concinnity of all this is an affected controller that has no process control integrity. Anyone with access to a network hosting an afflicted PLC can update the logic in the PLC, and can thus control the process[1]. In addition, it can allow an attacker to use an afflicted PLC as a persistent foothold on a process control network. In fact, malicious attackers may have already figured this out themselves.

In addition to the vulnerabilities found by the Reid Wightman, another researcher discovered other vulnerabilities[12]. This demonstrates independantly that the quality assurance teams at CoDeSys have failed to capture escaping security defects on multiple occasions. This is not uncommon because we know the cost of failure lies with their customers. Perhaps we can take refuge in 'Caveat emptor' then, as a protection against vulnerable systems?

Unfortunately, the defects appear to be inherited in over 200 types of products and devices. In a world where industrial system downtime can cost millions per hour[6], it would be expected that some supply chain quality assurance would have detected these vulnerabilities. However, to the best of the authors' knowledge only one company avoided these defects by using a threat modelling process. How can 200 product lines fail to establish that a password can be bypassed, allowing rogue ladder logic upload? To those less familiar with these systems, this is an equivalent of remote code execution from an unauthenticated attacker.

This then leaves us to assume that owners and operators of such vulnerable and important equipment would at least never place it on the open internet. Unfortunately, the authors were able to find roughly 600 vulnerable devices running directly on the open internet, as discussed in the next section.

[1] E.G. those that the authors found on the open internet

3 The Scan and The Incident Response

3.1 0.0.0.0/0 vulnerable CoDeSys device scan

Based on other works [9], the authors believed they would find some of these devices deployed (and thus exposed) on the open internet. Having determined that they would not crash a device with the scanning script, they set out to scann IPv4 and find out how many vulnerable controllers there were. To accomplish this task two linux machines were hosted in two different countries. The scan methodology consisted of two stages:

1. Using UnicornScan[7] to determine machines with either TCP port 1200 or 2455 open. Please see Figure 3
2. Using an NMAP NSE script to determine which of this subset was a vulnerable CoDeSys device. Please see Figure 4

The first task took about 6 months and the second about a week to complete. In total the project spent approximately $500 USD in hosting costs. Reserved blocks were ignored and people who contacted us with a cease and desist were removed from the blocks to be scanned. They were able to find us via a webpage hosted on the servers, if they detected the scans. The contact info page contained this text:

```
This server is conducting an internet survey of devices which run the
CoDeSys protocol.

The CoDeSys protocol is used in many industrial control applications
and contains serious vulnerabilities which could impact the integrity
of control systems which use it. The scan sends a harmless query for
the version of CoDeSys running, to verify that the IP is speaking the
CoDeSys protocol.

This protocol should not be internet-facing. This scan will determine
internet-facing controllers and note what their underlying operating
system is.

Findings will be shared with the appropriate IP block administrator
(if available) as well as the responsible CERT team (if available)
so that the controller can be secured.

If you'd like to be added to the blacklist from this scan, please
send an email to admin@example.com

The following IP address is used in the scan: xxx.xxx.xxx.xxx
```

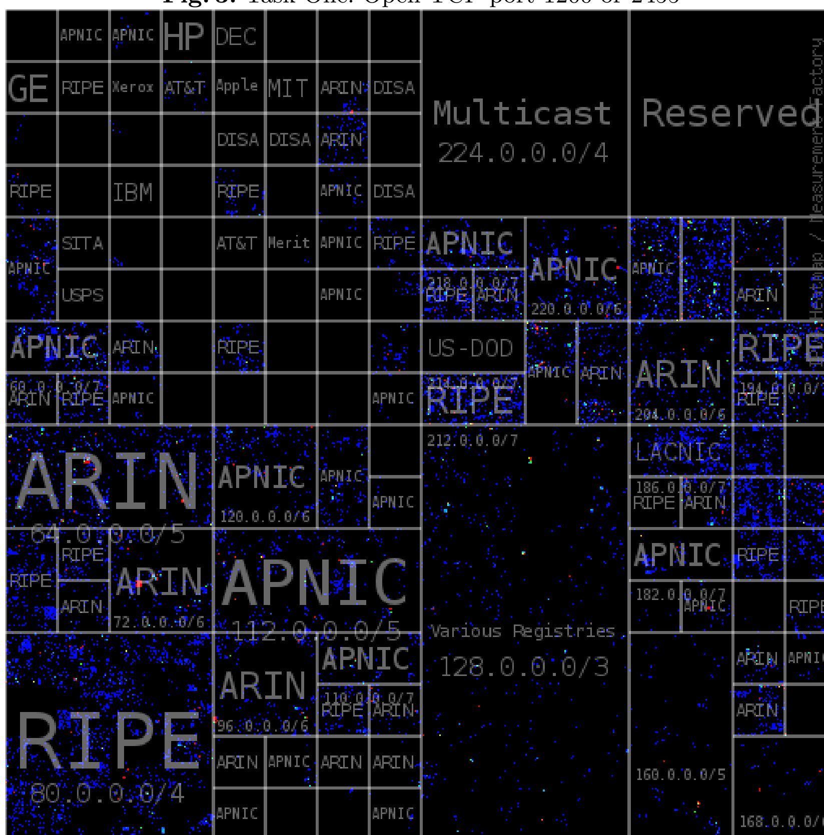
Amusingly, one organsiation asked us to stop scanning their network but couldn't tell us what their IP range was! Luckily, they understood the value of

our research and just asked to be informed if they had any vulnerable equipment. In total the researchers received 11 cease and desist letters while using a packet per second rate of 750. This means the majority of people do not detect scans at that rate, or are not interested in these ports. Again this underscores the vulnerability of industrial systems in a new way, with most organisations failing to detect reconnaissance on those vulnerable systems.

It will no doubt be asked why the authors did not choose to use ZMAP [13]. The answer is simply that this scan was conducted over the winter of 2012 and spring of 2013. At that point the ZMAP work cited had not been published. We will return to the appearance of ZMAP and its impact in a later section.

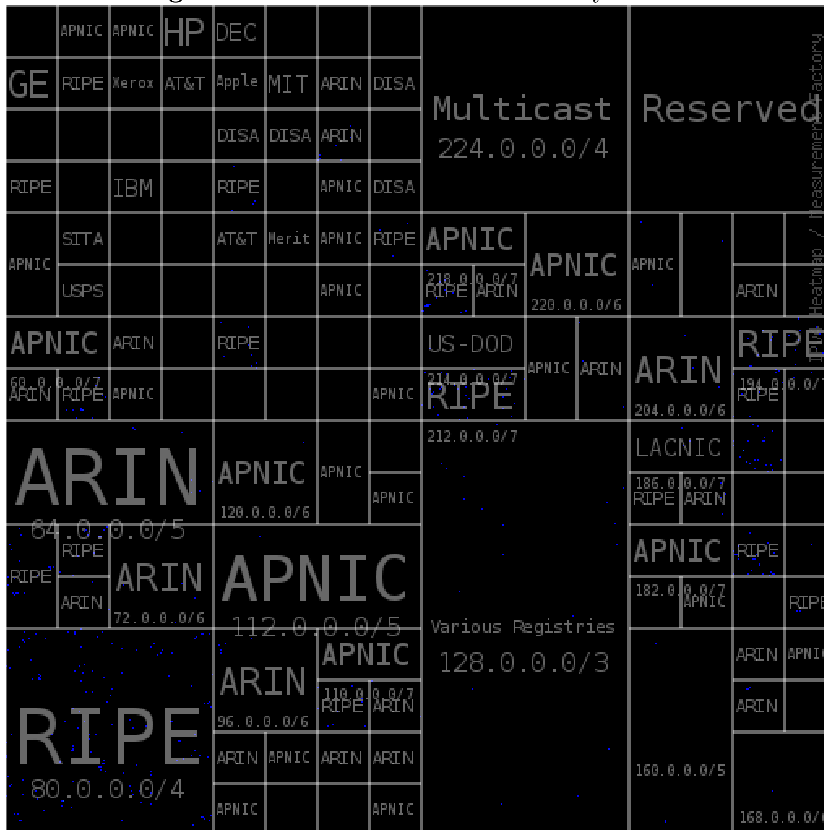
Below we provide two Hilbert Curve heatmap visualisations of the output of each stage. They were constructed using IPv4 Heatmap software available from the Measurement Factory[8]. Both images visualise one pixel as a /14 netblock. The authors expected a concentration to exist in certain netblocks. However, either the numbers are too small, or concentrations don't occur as we expected.

Fig. 3. Task One: Open TCP port 1200 or 2455



GreHack

Fig. 4. Task Two: 600 vulnerable CoDeSys devices



After analysing the results, it became clear that the bulk of vulnerable CoDeSys controllers existed primarily in RIPE3, with a smaller amount in ARIN space. The authors cannot account for this other than to suggest that these registries cover primarily industrialised nations. The Ten Autonomous System Numbers with the largest number of exposed PLCs is listed in Table 1.

In Table 2 we can see that the first ten countries account for 66% of the vulnerable CoDeSys systems found on the internet. This and the table above motivate our work sharing these vulnerable systems with CERTs who are interested in collaborating with us. It is clear that by working through them to contact vulnerable asset owners, we can make a substantial impact with just a few collaborations.

We know that not all of these systems will be classified as critical national infrastructure (CNI), however we have found that finding exposed industrial systems is a decent proxy towards protecting CNI. To put it concisely, CNI is often a subset of ICS/SCADA systems. More importantly, criticality effects can be produced by affecting a large number of non-critical devices. Consider for

GreHack

Table 1. Ten Autonomous Systems containing the largest number of vulnerable PLCs

PLCs Found	ASN	CC	Registrar	AS Name
9	6327	CA	arin	Shaw Communications Inc.
9	6830	AT	ripencc	Liberty Global Operations B.V.
12	5610	CZ	ripencc	Telefonica Czech Republic, a.s.
21	28929	IT	ripencc	ASDASD-AS ASDASD srl
25	12605	AT	ripencc	LIWEST Kabelmedien GmbH
28	3269	IT	ripencc	Telecom Italia S.p.a.
28	3303	CH	ripencc	Swisscom (Switzerland) Ltd
43	1136	EU	ripencc	KPN Internet Solutions[2]
43	286	EU	ripencc	KPN Internet Backbone
44	3320	DE	ripencc	Deutsche Telekom AG

example if an attacker decided to target all of the vulnerable devices within a single country.

Since Italy has the largest number in this particular study, let us use it as an example. We are left to wonder what the effect would be of a simple attack such as 81 industrial system devices ceasing to function at the same time. Obviously there are more subtle uses that require more reconnaisance and intelligence to create, an effect noted in this excellent paper[1].

Table 2. Ten Countries containing the largest number of vulnerable PLCs

PLCs Found	Country Code
21	CA
21	ES
29	CZ
33	AT
33	US
38	CH
60	PL
64	NL
80	DE
81	IT

4 How do such vulnerabilities bypass detection?

While these authentication bypass issues escaped the QA team at CoDeSys, it is not surprising or particularly uncommon. Software defects have been with us for a very long time. What is more surprising is that the companies integrating these libraries also did not detect such vulnerabilities. Do software and firmware teams relying on third party software simply integrate it without testing? Clearly,

Table 3. Number of Vunerable Devices per RIR

PLCs Found	Registry
0	afrnic
4	apnic
6	lacnic
54	arin
526	ripencc

the laudable approaches documented in scholarly articles are not reaching the engineering teams[5].

The most shocking thing to these researchers is that one year after an ICS-CERT alert went out to these companies, 600 vulnerable devices can be still be found on the open internet. Other work tells us that the average patch time in SCADA environments is 18 months[6]. However, we do find it surprising that vulnerable systems have not been removed from their internet exposure during that period. We can only assume either other mitigations are in place, or that the awareness campaign has yet to penetrate these corners of ICS device deployment.

5 Conclusion

Disregarding the researchers' time, the cost of this exercise is approximately .86 (USD) cents per vulnerable device found. Using a similar approach in the past the authors' were able to find devices at a cost of roughly $1.56 per device. We aim to motivate other researchers to adopt the same approach and log their cost per device as a future metric. Why would this metric be interesting or novel?

There are 9 reasons:

1. This metric is methodology and technology independant.
2. As costs for parallelisation fall this is incorporated into the metric.
3. As newer, faster scanners (such as ZMAP) are developed this is also included in the metric.
4. The density of vulnerability across a network space is factored into the metric.
5. Partial scans can still be used for metrics.
6. We understand the cost to attackers of finding opportunistic targets.
7. We understand the low cost to this methodology of defending.
8. We understand the change over time in the lifecycle of exposure and vulnerability.
9. It naturally translates a technical problem into an economic one ready for debate and policy discussion.

Let us test the assumptions of these metrics by extrapolating the results onto a theoretical use of ZMAP. In early 2012 it cost the authors of this paper .86 cents per vulnerable CoDeSys device. If they had used ZMAP instead, but the same hardware and approach the cost would have been .11 cents per device. For

clarity, that's 2 machines at 35 per month, with the scans completing in a single week. The authors would not scan as quickly as possible, because they believe this generates more trouble for sysadmins, and more cease and desist letters for themselves (thus a week).

This clarifies the falling cost of reconnaisance for attackers, and makes it clear that defenders could benefit from this approach if we remove legal and administrative barriers to doing so. We believe this metric is the primary contribution of this paper, and the NSE script for finding vulnerable CoDeSys systems a secondary contribution. The rest of the paper is simply a published record of the authors' efforts to help secure the industrial systems of any country in the world over a six month period.

Considering the riduculously low cost of finding these vulnerable devices, why hasn't CoDeSys done this work themselves? This suggests market failure for ICS security to the authors, although they freely admit they are not economists. Perhaps we can motivate an economist to examine this case study and publish their findings. By publishing our costs, we hope this becomes possible in the future.

The authors think this is an incredibly cheap approach to helping nations secure their industrial control infrastructure, and as a side effect, verify and enhance the protection of their critical national infrastructure. If it sounds crazy to spend a portion of the money allocated to cyber defense scanning for vulnerabilities, we would like to point the reader toward excellent work in the security economics of malware mitigation at scale[4][3]. Where the compromise of industrial infrastructure has the potential to levy a heavy cost incident to society, isn't scanning and remediating through incident response teams one of the quietist and most cost effective ways of reducing national attack surfaces?

References

1. Rid, Thomas. "Cyber War Will Not Take Place" Journal of Strategic Studies, (2012)
2. Ladder Logic, http://en.wikipedia.org/wiki/Ladder_logic
3. Hofmeyr, Steven, et al. "Modeling internet-scale policies for cleaning up malware." Economics of Information Security and Privacy III. Springer New York, (2013)
4. Clayton, Richard. "Might Governments Clean-up Malware?." Communication and Strategies 81 (2011)
5. Reichenbach, Frank, et al. "A Pragmatic Approach on Combined Safety and Security Risk Analysis." Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on. IEEE, (2012)
6. Eric Byres and Justin Lowe. The Myths and Facts behind Cyber Security Risks for Industrial Control Systems. Proceedings of the VDE Kongress, (2004)
7. UnicornScan: Unicorns are fast! www.unicornscan.org/
8. IPv4 Heatmap Software http://maps.measurement-factory.com/software/index.html
9. Leverett, Eireann P: Quantitatively Assessing and Visualising Industrial System Attack Surfaces. MPhil Thesis, University of Cambridge Darwin College (2011)

10. N3S-Software CoDeSys Improper Access Control (Update), http://ics-cert.us-cert.gov/alerts/ICS-ALERT-12-097-02A
11. 3S CoDeSys Multiple Vulnerabilities, http://ics-cert.us-cert.gov/advisories/ICSA-13-011-01
12. 3S CODESYS Gateway-Server Multiple Vulnerabilities (Update A), http://ics-cert.us-cert.gov/advisories/ICSA-13-050-01A
13. Zakir Durumeric and Eric Wustrow and J. Alex Halderman: ZMap: Fast Internet-Wide Scanning and its Security Applications. Proceedings of the 22nd USENIX Security Symposium, (2013)
14. Beck IPC GmbH sc1x3 CoDeSys Processors, http://www.beck-ipc.com/en/products/sc1x3/index.asp

Code Appendix

NMAP NSE script for determing vulnerable CoDeSys devices

```
description = [[ development
author = "hdm"
-- minor tweaks and bugfix by krw
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"discovery", "safe"}
local nmap   = require "nmap"
local comm   = require "comm"
local stdnse = require "stdnse"
local strbuf = require "strbuf"
local nsedebug = require "nsedebug"

-- Script is executed for any TCP port.
portrule = function( host, port )
  return port.protocol == "tcp"
-- Grabs a banner and outputs it nicely formatted.
action = function( host, port )
  local out = grab_banner(host, port, "\187\187\001\000\000\000\001")
  if out == "EOF" then
    -- try a big-endian query
    out = grab_banner(host, port, "\187\187\000\000\000\001\001")
  end
  return output( out )

-- Returns a number of milliseconds for use as a socket timeout
-- value (defaults to 5 seconds).
-- @return Number of milliseconds.
function get_timeout()
  return 5000

-- Connects to the target on the given port and returns any
```

```
-- data issued by a listening service.
-- @param host  Host Table.
-- @param port  Port Table.
-- @return      Socket descriptor and initial banner
function grab_banner(host, port, query)
  local st, buff, banner
  local proto  = "tcp"
  local socket = nmap.new_socket()
  socket:set_timeout(get_timeout())
  banner = ""
  proto = "tcp"
  st = socket:connect(host, port, proto)
  if not st then
    socket:close()
    return nil
  end
  socket:send(query)
-- Big endian version
-- socket:send("\187\187\000\001\000\000\001")
-- socket:send("\xbb\xbb\x01\x00\x00\x00\x01")
  st,banner = socket:receive()
  socket:close()
  return banner

-- Formats the banner for printing to the port script result.
-- Non-printable characters are hex encoded and the banner is
-- then truncated to fit into the number of lines of output desired.
-- @param out  String banner issued by a listening service.
-- @return     String formatted for output.
-- Ripped from banner.nse with line wrap disabled (corrupts output)
function output( out )
  if type(out) ~= "string" or out == "" then return nil end
  local filename = SCRIPT_NAME
  local line_len = 75
  -- The character width of command/shell prompt window.
  local fline_offset = 5
  -- number of chars excluding script id not available to the script
  -- on the first line
  -- number of chars available on the first line of output
  -- we'll skip the first line of output if the filename is looong
  local fline_len
  if filename:len() < (line_len-fline_offset) then
    fline_len = line_len -1 -filename:len() -fline_offset
  else
    fline_len = 0
```

```
  end
  -- number of chars allowed on subsequent lines
  local sline_len = line_len -1 -(fline_offset-2)
  -- replace non-printable ascii chars - no need to do the whole string
  out = replace_nonprint(out, (out:len() * 3) + 1)
  -- 1 extra char so we can truncate below.
  -- break into lines - this will look awful if line_len is more than
  -- the actual space available on a line...
  local ptr = fline_len
  local t = {}
  t[#t+1] = out
  return table.concat(t,"\n")

-- Replaces characters with ASCII values outside of the range of standard printable
-- characters (decimal 32 to 126 inclusive) with hex encoded equivalents.
-- The second parameter dictates the number of characters to return, however, if the
-- last character before the number is reached is one that needs replacing then up to
-- three characters more than this number may be returned.
-- If the second parameter is nil, no limit is applied to the number of characters
-- that may be returned.
-- @param s    String on which to perform substitutions.
-- @param len  Number of characters to return.
-- @return     String.
-- Pulled from banner.nse and mangled to escape \r\t\n separately
function replace_nonprint( s, len )
  local t = {}
  local count = 0
  for c in s:gmatch(".") do
    if c:byte() == 9 then
      t[#t+1] = ("\\%s"):format("t")
      count = count+3
    elseif c:byte() == 10 then
      t[#t+1] = ("\\%s"):format("n")
      count = count+3
    elseif c:byte() == 13 then
      t[#t+1] = ("\\%s"):format("r")
      count = count+3
    elseif c:byte() < 32 or c:byte() > 126 then
      t[#t+1]=("\\x%s"):format(("0%s"):format(((stdnse.tohex(c:byte())):upper())):sub(-2
      -- capiche
      count = count+4
    else
      t[#t+1] = c
      count = count+1
    end
```

```
    if type(len) == "number" and count >= len then break end
  end
  return table.concat(t)
```

(Script Written by hdmoore and altered and adapted by Reid Wightman)