REXX/Sockets Update

ARTHUR ECOCK

ARTY.ECOCK@GMAIL.COM

Prelude

- Why do this?
  - Request from Tim Kessler for 5 TLS IOCTLs – 2018
  - Perry and Dave gave TLS presentation at 2019 VM Workshop
  - "Heads up" from Tim for new IOCTL – 2020
- So, a little recent interest
- Really no room in code to do anything significant
- IPv6 gaining my interest for past several years
- More involvement with TLS

Agenda

- Base code changes
- TLS changes
- IPv6 changes

Brief History

- REXX/Sockets is a REXX function package that allows client/server communication using BSD-style "sockets"
- Much like the low-level communication protocols that power the Internet
- REXX/Sockets "exposes" an Assembler-based IUCV API to the REXX language
- Focus on keeping function syntax "native" to REXX
  - C: connect(*s*, **name*, *namelen*);
  - REXX: Socket('Connect', s, 'AF_INET 443 www.site.info')
- Many Socket('…') functions

Base code changes (the past)

- Original code used 3 (4) base registers
- Re-shuffled subroutines and data areas for additional addressability
- Re-coded to use address constants (yuch)
- Added external code (PL/X, yuch)
- Not much addressability left
  - Difficult to add new function (SIOCGCERTDATA)

Base code changes (the present)

- Perry Ruiter to the rescue
  - XEDIT magic to change all Branch instructions to Branch Relative instructions
- Found a paper by Sharuff Morsa (IBM UK)
  - "Relative Addressing (what to do when you run out of base registers)"
    - Replace L rx,=A(label) with LARL rx,label
- With these resources, initial goal was to:
  - Replace short labels with more readable (longer) labels (maintainability)
  - Use 1 base register, not 4
  - Refactor a few routines to reduce code redundancy
  - Use "immediate" and "relative" instructions where appropriate
  - Fix a few bugs
  - Clean up "existing" TLS code

Base code changes

- No further addressability issues!
- Room for more code
- Lots of new code:
  - TLS IOCTLs
  - IPv6
  - Much better DNS support
  - Enhanced diagnostics

Base code changes

- Socket('Version')

  0 REXX/SOCKETS 3.05 12 April 1996

- Version moved from 3.04 to 3.05 (incremental change)

TLS changes

- SIOCTLSQUERY
- SIOCSECCLIENT
- SIOCSECSERVER
- SIOCSECSTATUS
- SIOCSECCLOSE
- SIOCGCERTDATA

TLS changes

- SIOCTLSQUERY
  - Socket('IOCTL', socket, 'SIOCTLSQUERY', label)
  - rc = 0 indicates the TLS "label" is present in the gskkyman database, and an SSL server Is available
  - If "label" was blank, rc = 0 indicates an available SSL server

TLS changes

- SIOCSECCLIENT
  - Socket('IOCTL', socket, 'SIOCSECCLIENT', label, <options>, <msg>)
  - Socket('IOCTL', socket, 'SIOCSECCLIENT', label,
    'NoSSLv2 fqdn=server.com hostname=server Alert')

  Options:
    - NoSSLv2: "Don't use SSLv2 ciphers"
    - fqdn, hostname: "Check server certificate fields for matching data"
      - ipv4 and ipv6 keywords also supported
      - These options imply "Host Validation is Required"
    - Alert: "Don't fail the SSL Handshake if fields don't match, just inform me"

TLS changes

- SIOCSECSERVER
  - Socket('IOCTL', socket, 'SIOCSECSERVER', label, <options>, <msg>)
  - Socket('IOCTL', socket, 'SIOCSECSERVER', label,

    'NoSSLv2 ClientCertCheck=Required')

  Options:
    - NoSSLv2: "Don't use SSLv2 ciphers"
    - ClientCertCheck: "Request client certificate and validate"
      - ClientCertCheck=None (default) and ClientCertCheck=Preferred also supported
    - ~~Other options supported: NoCheck (default), FullCheck, ValidatePeerCert, RequestClientCert, SSLv2 (however, NoCheck, FullCheck, and RequestClientCert must be used in specific combinations to achieve Client Certificate Checking: Use ClientCertCheck option instead)~~

TLS changes

- SIOCSECSTATUS
  - Socket('IOCTL', socket, 'SIOCSECSTATUS')
    - 0 SecDynamic TLS12 SHA1 DES3 ECDHE_RSA 168
    - rc: 0
    - Security Level: SecNone | SecStatic | SecDynamic
    - Cipher Class: Null | SSLv2 | SSLv3 | TLS | TLS10 | TLS11 | TLS12
    - Cipher Hash: Null | MD5 | SHA1 | SHA2 | SHA256 | SHA384
    - Cipher Algorithm: Null | RC2 | RC4 | DES | FIPSDES | FIPS3DES | AES | AESGCM | AES128 | AES128GCM | AES256 | AES256GCM
    - Cipher PK Algorithm: Null | RSA | DH_DSS | DH_RSA | DHE_DSS | DHE_RSA | ECDH_ECDSA | ECDHE_ECDSA | ECDH_RSA | ECDHE_RSA
    - Cipher Key Length: integer

TLS changes

- SIOCSECCLOSE
  - Socket('IOCTL', socket, 'SIOCSECCLOSE', <msg>)
  - If "msg" is specified, data pipeline is flushed and replaced with "msg" string

TLS changes

- SIOCGCERTDATA
- Socket('IOCTL', socket, 'SIOCGCERTDATA', <side>, <codes>)
- Socket('IOCTL', socket, 'SIOCGCERTDATA', 'partner',
  'CN OU O L ST C')

0 6 CN='maint710.company.com' OU='Office' O='Corporate' L='Freehold' ST='New Jersey' C='US'

- Socket('IOCTL', socket, 'SIOCGCERTDATA', 'partner', 'DN')

0 1 DN='CN=maint710.company.com,OU=Office,O=Corporate,L=Freehold,ST=New Jersey,C=US'

TLS changes

- SIOCGCERTDATA
  - side: local or partner
  - codes: cert_body_der, cert_body_base64, cert_serial_number, cn, cert_common_name, l, cert_locality, st, cert_state_or_province, c, cert_country, o, cert_org, ou, cert_org_unit, dn, cert_dn_printable, cert_dn_der, cert_postal_code, email, cert_email, cert_domain_component, sn, cert_surname, cert_street, cert_title, cert_issuer_common_name, cert_issuer_locality, cert_issuer_state_or_province, cert_issuer_country, cert_issuer_org, cert_issuer_org_unit, cert_issuer_dn_printable, cert_issuer_dn_der, cert_issuer_postal_code, cert_issuer_email, cert_issuer_domain_component, cert_issuer_surname, cert_issuer_street, cert_issuer_title, cert_name, cert_givenname, cert_initials, cert_generationqualifier, cert_dnqualifier, cert_mail, cert_serialnumber, cert_issuer_name, cert_issuer_givenname, cert_issuer_initials, cert_issuer_generationqualifer, cert_issuer_dnqualifier, cert_issuer_mail, cert_issuer_serialnumber

IPv6 “Addressing 101”

- IPv6 addresses are 128 bits (16 bytes) (8 hextets)
- 2001::bad:c0ff:ee:bad:code
- 340 Undecillion addresses
- 340 Billion, Billion, Billion, Billion addresses
- ~ 6 Octillion addresses per person on Earth

- IPv4 addresses are 32 bits (4 bytes) (4 octets)
- 123.123.123.123
- ~ 4 Billion addresses
- <1 address per person on Earth

IPv6 changes

- Full IPv6 support
  - Seamless
    - Old code still works (no changes)
    - Old code can leverage IPv6 DNS servers
    - IPv6 additions non-invasive
  - Full DNS support
    - More friendly diagnostics
  - Mixed IPv4/IPv6 sockets
  - IUCV API icky-ness hidden
    - 2 separate IUCV APIs, 1 socket abstraction

IPv6 changes

- Goal was to preserve REXX/Sockets API
  - Create IPv4 or IPv6 sockets within the same socketset
- "AF_INET" already present in API, so adding "AF_INET6" is fair game
- Needed to extend existing API just a little to allow "AF_INET6" domain to be specified where appropriate
  - Socket('Resolve', 'www.facebook.com') ← Returns IPv4 address
  - Socket('Resolve', 'www.facebook.com', 'AF_INET6') ← Returns IPv6 address
  - Socket('Resolve', 'www.facebook.com', 'AAAA') ← Returns IPv6 address

IPv6 changes

- Deviated from z/OS REXX/Sockets:
  - Socket "name" is:
  - domain port ipaddress (Family Port Address)
  - AF_INET 1234 xxx.xxx.xxx.xxx (3 fields)
  - AF_INET6 1234 <flowid> xxxx:xxxx:xxxx:: <scopeid> (5 fields) (yuch)
- "flowid" not supported by z/VM API
- "scopeid" is barely supported
- Decided to use:
  - AF_INET6 1234 xxxx:xxxx:xxxx:: (3 fields) (yay)

IPv6 changes

- IUCV API Type “4”
  - Provides IPv6-only sockets
  - Is there value keeping socketsets restricted to a single Address Family?
- Nope, not “natural”, not the BSD way
- Decided to “hide” the fact that all IPv6 sockets needed to be driven using a separate IUCV path/API
- REXX Programmer simply creates IPv4 and/or IPv6 sockets and uses them as intended (mix-and-match)
- No concern for the underlying API mess

IPv6 changes

- Better programming experience, but trickier to implement
  - Socket('Select', 'Read 1 2 3 4 Write 1 2', timeout)
  - If socket 1 is IPv4 and socket 2 is IPv6, underlying code needs to drive 2 Select calls (1 through the API Type 3 path and 1 through the API Type 4 path)
  - Whichever call finishes first needs to Cancel the other

  - Socket('Select', mask, 'IDENTIFY')
  - If mask contained a mix of IPv4 and IPv6 sockets, which "messageID" should be returned? (2 separate IUCV APIs, so 2 messageIDs)
  - In such cases, I tie the 2 requests together and treat them as 1 (externally)

IPv6 changes

- Socket('Cancel', messageID)
  - Does the "right" thing
  - No changes required to API (nor REXX programs)
- Socket('Select', mask, options)
  - Does the "right" thing
  - No changes required to API (nor REXX programs)

IPv6 changes

- So, what *does* change in the API?
- A few extensions, but the API is stable
  - family4 = "AF_INET"
  - family6 = "AF_INET6"
  - Socket('Socket') ← Default is "AF_INET", as usual
  - Socket('Socket', family4) ← "domain" may be specified, as usual
  - Socket('Socket', family6) ← "AF_INET6" is a new domain

  - Socket('Bind', socket, family4 port 'inaddr_any') ←IPv4 domain & address
  - Socket('Bind', socket, family6 port 'in6ddr_any') ←IPv6 domain & address

IPv6 changes

- IPv6 address can look like:
- Colon-delimited hextets:
  - 2001:01db:dead:beef:cafe:feed:bad:f00d
  - ::1
  - ::
  - 2001:1db:bad:beef::
  - 2001:1db:6464::128.228.1.2 ← Yes, that's valid
  - ::ffff:128.228.1.2 ← "Mapped" addresses, too

IPv6 changes

- Socket('GetClientId') ← "AF_INET" still the default
- Socket('GetClientId', family4) ← "domain" may be specified
- Socket('GetClientId', family6) ← "AF_INET6" is a new domain

- Socket('GetSockName', socket) ← IPv4 *or* IPv6 address returned
- Socket('GetPeerName', socket)
- Socket('Accept', socket)
- Socket('RecvFrom', …)

- Socket('Bind', …) ← IPv4 *or* IPv6 address specified
- Socket('Connect', …)
- Socket('SendTo', …)

IPv6 changes

"domain" influences the behavior of:

- Socket('GetHostByAddr', …)
- Socket('GetHostByName', …)
- Socket('Resolve', …)

IPv6 changes

- Socket('Trace', 'Resolver')

Checking DNS servers for stanford.edu

Connecting to NameServer: 8.8.8.8, Time: 13:58:24

Question to NameServer: 8.8.8.8, Time: 13:58:24, ResolverTimeout: 5 seconds

001E0001 01000001 00000000 00000873 74616E66 6F726403 65647500 00FF0001

Answer from NameServer: 8.8.8.8, Time: 13:58:24

102B0001 81800001 00280000 00000873 74616E66 6F726403 65647500 00FF0001

C00C002E 00010000 070700A0 001C0802 00000708 5FF83775 5FD09FFD 26D30873

- Displaying the Question and Answer is nice, but we can do better than hexadecimal
- Noticed quite a few DNS server-related issues, so better diagnostics were warranted

IPv6 changes

- Socket('Trace', 'Resolver')
  - New diagnostic message to partially interpret Answer:

Flags: qr rd ra aa (8580); Answer: 6, Authority: 0, Additional: 0

- Helpful, but sometimes deciphering the Answer section is necessary
  - "I received 6 Answers to my DNS Query, but none of them seem to be the one I was seeking, so what *were* the Answers?"
  - (In one of my test cases, I was seeking an AAAA record from a DNS server and I was receiving a troubling response. I needed more data.)

IPv6 changes

- Socket('Trace', 'Resolver ANY')

Flags: qr rd ra aa (8580); Answer: 6, Authority: 0, Additional: 0

>?> cuny.edu, type = ANY, class = IN

>A> cuny.edu. 3600 NS d-395h-5-dcdns-2.cis.

>A> cuny.edu. 3600 MX 10 mail-relay.cuny.edu.

>A> cuny.edu. 3600 A 172.18.192.200

>A> cuny.edu. 3600 TXT "v=spf1 ip4:128.228.0.167 ..."

>A> cuny.edu. 3600 NAPTR (not formatted)

>A> cuny.edu. 3600 SOA 555w-dnsco.cuny.edu. ...

- Ok, no AAAA Answer returned, Authoritative Answer ("aa"), clear as day: no IPv6 address. (In my test case, this was due to lack of IPv6 support in the Name Server.)

IPv6 changes

- Socket('Trace', 'Resolver Any')
- Socket('Resolve', 'Stanford.edu') ← I used 8.8.8.8 for NSinterAddr

Flags: qr rd ra (8180); Answer: 40, Authority: 0, Additional: 0

>?> stanford.edu, type = ANY, class = IN

>A> stanford.edu. 1799 RRSIG (not formatted)

>A> stanford.edu. 1799 AAAA 2607:f6d0:0:925a::ab43:d7c8

>A> stanford.edu. 21599 SOA argus.stanford.edu. hostmaster.

stanford.edu. 2020188159 1200 600 1296000 1800

… so many more Answers

IPv6 changes

- Socket('Translate', …)
  - "To_IPv6_Address"
    - Convert 16-byte hexadecimal IPv6 to (Printable) character format ("ntop")
    - Convert Printable IPv6 address to 16-byte hexadecimal format. ("pton")
      - Hex: FF020000000000000000000000000001
      - Char: ff02::1
  - "To_SockAddr_In6"
    - Convert 28-byte hexadecimal sockaddr_in6 to Printable "Name"
    - Convert Printable "Name" to 28-byte hexadecimal format sockaddr_in6
      - Hex: 001300000000000000000000000000000000000000000100000000
      - Char: AF_INET6 0 ::1

IPv6 changes

- IPv6 addresses supported in:
  - ETC HOSTS
    - As a result, we can eliminate the PL/X code (DMSRXR)
  - TCPIP DATA
    - Full support for IPv6 Name Servers

IPv6 changes

- GetAddrInfo and GetNameInfo were *not* added
  - Not convinced they are needed
  - z/OS sample:

```
Socket("GetAddrInfo","MVS150",54777,
        "AI_ALL AI_CANONNAMEOK AI_NUMERICSERV AI_V4MAPPED",
        "AF_INET6","SOCK_STREAM","IPPROTO_TCP");
```

  - Yuch
  - Convince me otherwise (uh, see next slide...)

IPv6 Changes

- One look at a complicated application and I changed my mind and coded Socket('GetAddrInfo',…)
- Pared down, it's quite useful (easily code AF-agnostic programs)
  - A client can easily establish IPv6 *or* IPv4 connections to a target server depending on the target's DNS records
  - Client: Socket('GetAddrInfo', server, port, 'AF_UNSPEC')
  - Returns: rc fqdn name name name name …
    - "name": AF_INETx 12345 IP_address
  - Loop on names, trying to establish a Connection
  - Server: Socket('GetAddrInfo', , port, 'AF_UNSPEC AI_PASSIVE')
  - Server then feeds "name" data to a Socket/Bind/Listen combo
- Socket('GetNameInfo',…) also added for symmetry

GetAddrInfo

- Recognized the value of this function call
  - Value: Address Family agnostic code (AF_INET, AF_INET6)
- Implemented without socket "Type" and "Protocol" options
  - These options were of limited value
- AI_ADDRCONFIG option not implemented
  - Too much controversy over correct implementation
    - "Avoid DNS lookups" versus "Return IPx information only if IPx address is configured on an interface"
  - Limited value
  - Stack really doesn't care (V4-mapped addresses)
- Slim version of GetAddrInfo quite elegant

Prior Client/Server scenarios

- Socket('Initialize', …)
- Socket('Socket', 'AF_INET', …) /* Address Family "hard-coded" */
- /* "NAME" triplet completely specified: family port address */
- Socket('Connect', socket, 'AF_INET 5678 123.123.123.123')

- Socket('Initialize', …)
- Socket('Socket', 'AF_INET', …)
- /* "Bind" address (passive, loopback or IP) "hard-coded" */
- Socket('Bind', socket, 'AF_INET 5678 123.123.123.123')
- Socket('Listen', socket, 10)
- Socket('Select', …)
- Socket('Accept')

GetAddrInfo (Client Scenario)

- A client *may* wish to connect to a server, preferring its IPv6 address if available, but use IPv4 address otherwise
- We'll use GetAddrInfo with the AF_UNSPEC option to request both IPv6 and IPv4 information for a given server/port combination (port is optional, of course)
- Specifying port causes the port to appear in the results

GetAddrInfo (Client Scenario)


```
Parse Value Socket('GetAddrInfo', server, port, 'AF_UNSPEC') With rc fqdn names
connected = 0
If rc=0 Then Do Until connected
  If names="" Then Exit 1

  /* Parse each "NAME" triplet from GetAddrInfo */
  Parse Var names ai_family ai_port ai_address names

  /* Attempt to create a socket with the correct Address Family */
  Parse Value Socket('Socket', ai_family) With rc socket .
  If rc=0 Then Do
```

GetAddrInfo (Client Scenario)


```
 /* Socket creation successful, now try a connect */
Parse Value Socket('Connect', socket, ai_family ai_port ai_address) With rc .

If rc=0 Then connected = 1
Else rc = Socket('Close', socket)
  End
End
```

GetAddrInfo (Server Scenario)

- A server may wish to offer services on both IPv6 and IPv4 interfaces
- We'll use GetAddrInfo with the AF_UNSPEC option to request both IPv6 and IPv4 information
- We'll omit the server (host) parameter, since we only want our "0" addresses (::0 and 0.0.0.0) - "bind" to ANY IP address
  - Other examples may wish the server to Bind to specific IP addresses
- We specify the port, because we want it to appear in the results
- We also use the AI_PASSIVE option because we are interested in using the results in a subsequent Bind operation (without AI_PASSIVE, the loopback addresses ::1 and 127.0.0.1 would be returned, since we are omitting the host parameter)

GetAddrInfo (Server Scenario)


```
Parse Value Socket('GetAddrInfo', , port, 'AF_UNSPEC AI_PASSIVE') With rc . names

Do While names<>""
  Parse Var names ai_family ai_port ai_address names
  If ai_family<>"" Then Do
    Parse Value Socket('Socket', ai_family) With rc socket .
    If rc=0 Then Do
      rc = Socket('Bind', socket, ai_family ai_port ai_address)
      If rc=0 Then Do
        Say "Socket" socket "bound to" ai_address".."ai_port
        Leave
      End
    End
  End
End                                    /* "Normal" program logic follows ... */
rc = Socket('Listen', socket, '10')
```

GetAddrInfo (Client, Server, IPv6)

- After using GetAddrInfo and the snippets of code above, the remainder of the client and server code is unchanged
- GetAddrInfo makes it a little easier to "IPv6 enable" existing code
  - AI_V4MAPPED option serves as a nice bridge
- IP address checking/manipulation will need to be re-visited, however
  - Socket('Resolve', …) and Socket('Translate', …) may offer relief
  - (The thought here would be to use Resolve to translate IPv4 *or* IPv6 addresses to FQDNs, then use FQDNs for authorization/authentication checking, instead of messing with IP addresses)

GetAddrInfo

- AI_V4MAPPED option allows translation of IPv4 addresses (and DNS results) into IPv6 “mapped” addresses:

Socket('GetAddrInfo','10.27.1.12', 443, 'AI_CANONNAME AI_V4MAPPED AF_INET6’)

"0 server.company.com AF_INET6 443 ::ffff:10.27.1.12"

- Resolve can be used to “un-map” an address:

Socket(‘Resolve’, ‘::ffff:10.27.1.12’)

“0 10.27.1.12 SERVER.COMPANY.COM” ← Yeah, it’s uppercase (sorry)

Fun with Translate and IPv6

- Use Socket('Translate') to convert an IPv6 address in non-standard, or un-compacted format into "Canonical" form (RFC5952)

```
Parse Value Socket('Translate','ff02:0:0:00:000::1','To_IPv6_Address') With rc len hex_IP
Parse Value Socket('Translate', hex_IP, 'To_IPv6_Address') With rc len char_IP
Say char_IP
ff02::1
```

- Same technique can be used for IPv4 addresses (use 'To_IPv4_Address')

Adding TLS (Client and Server sides)


```
/* Determine if TLS label (and an SSL server) is available */
Parse Value Socket('IOCTL', socket, 'SIOCTLSQUERY', tls_label) ,
   With rc errno text

If rc<>0 Then TLS=0
Else TLS = 1
```

(Note: Errors from SYSTEMSSL are included with descriptive text; rc=40xxx)

Adding TLS (Client side)


```
/* Connect was successful, now try to negotiate TLS */
If TLS Then Do
   options = 'NoSSLv2'
   /* options = 'NoSSLv2 fqdn=fred.com FullCheck ipv4=0.0.0.0 Alert' */
   Parse Value Socket('IOCTL', socket, 'SIOCSECCLIENT', tls_label, options) ,
      With rc .

   If rc<>0 Then Say TcpError('SIOCSECCLIENT')

   Parse Value Socket('IOCTL', socket, 'SIOCSECSTATUS') With rc setting type .
   Say "TLS setting is:" setting
End
```

Adding TLS (Server side)


```
/* Accept has completed and returned "new_socket", try TLS now */
If TLS Then Do

  Parse Value Socket('IOCTL', new_socket, 'SIOCSECSERVER', ,
    tls_label, 'ClientCertCheck=Preferred') With rc errno text
  If rc<>0 Then Say "TLS handshake failed:" text

  Parse Value Socket('IOCTL', new_socket, 'SIOCSECSTATUS') ,
    With rc setting .
  Say "TLS setting is:" setting
End
```

Adding TLS (Client and Server sides)


```
If setting="SecDynamic" Then Do          /* Or: If setting<>"SecNone" Then Do */
   Parse Value Socket('Ioctl', socket, 'SIOCGCERTDATA', 'partner', 'DN') With rc count dn

   If count=1 Then Say "Partner DN:" dn
End
```

Note: "DN" includes "CN" data (CN is typically FQDN, so it may be more useful)

Adding TLS

- Just 2 or 3 Socket() calls to add to client and server
- Certificate management remains the only challenge

REXX/Sockets update

- TLS update – 260 lines (2018)
- New TLS IOCTLs + IPv6 update – 12,000 lines (2020)

- Major success was removing the addressability issue, thus paving the way for future updates
  - Like: Fixing the Mutex issue once and for all!

- Amazing what a little spare time and incentive will do!

Thank you!!!