SIEM Tutorial

SOC Analyst ဆိုတာ Security Operation Center မှာ လုံခြုံရေးအဖြစ်အပျက်တွေကို စောင့်ကြည့်၊ စစ်ဆေး၊ ခွဲခြမ်းသုံးသပ်တဲ့ လူတွေဖြစ်ပြီး Log Data တွေအများကြီးကို နေ့စဉ် ကိုင်တွယ်ရပါတယ်။ ဒီလို Log Data များတဲ့အခါ အရေးပါတဲ့ အချက်အလက်ကို မြန်မြန်နဲ့ တိတိကျကျ ရှာဖွေနိုင်ဖို့ Query Language တစ်ခုလိုအပ်လာပါတယ်။ အဲဒီအတွက်ပဲ KQL(Kusto Query Language) ကို အသုံးပြုရတာ ဖြစ်ပါတယ်။

ဒီသင်ခန်းစာကို လေ့လာပြီးရင် KQL ကို အသုံးပြုပြီး Security Log Data တွေအတွင်းမှာရှိတဲ့ အချက်အလက်တွေကို ဘယ်လိုရှာရမလဲ၊ ရလာတဲ့ Data ကို ဘယ်လို နားလည်ဖတ်ရှုရမလဲ ဆိုတာကို သိလာပါမယ်။ Log Data ဆိုတာက အဖြစ်အပျက်တွေကို စနစ်တကျ မှတ်တမ်းတင်ထားတဲ့ စာရင်းတွေဖြစ်ပြီး တစ်ခါတစ်ရံ အရမ်းရှုပ်ထွေးပြီး မလိုအပ်တဲ့ အချက်အလက်တွေ အများကြီးပါဝင်နေတတ်ပါတယ်။ KQL ကို သုံးရင် အဲဒီ Data တွေအထဲကနေ လိုအပ်တဲ့ အချက်အလက်ကိုပဲ ရွေးချယ်ပြီး ကြည့်ရှုနိုင်ပါတယ်။

ထို့အပြင် Event Data တွေကို Filter လုပ်တာ၊ Parse လုပ်တာ၊ Aggregate လုပ်တာနဲ့ Transform လုပ်တာတွေကိုလည်း သင်ပေးထားပါတယ်။ Filter လုပ်တယ်ဆိုတာက ကိုယ်မလိုအပ်တဲ့ Event တွေကို ဖယ်ရှားပြီး လိုအပ်တဲ့ Event တွေပဲ ကျန်အောင် လုပ်တာဖြစ်ပါတယ်။ Parse လုပ်တယ်ဆိုတာက Log ထဲက စာကြောင်းရှည်တွေကို အဓိပ္ပါယ်ရှိတဲ့ အပိုင်းလေးတွေ ခွဲထုတ်တာပါ။ Aggregate လုပ်တယ်ဆိုတာက Event အရေအတွက်တွေကို စုစည်းပြီး နာရီလို၊ နေ့လို စုတွက်တာမျိုးဖြစ်ပါတယ်။ Transform လုပ်တယ်ဆိုတာက Data ပုံစံကို ပြောင်းလဲပြီး ပိုပြီး နားလည်လွယ်အောင် ပြင်ဆင်တာကို ဆိုလိုပါတယ်။ ဒီလုပ်ဆောင်ချက်တွေကြောင့် ပုံမှန်ကြည့်ရင် မမြင်ရတဲ့ Pattern တွေ၊ သံသယဖြစ်စရာ လက္ခဏာတွေကို ဖော်ထုတ်နိုင်လာပါတယ်။

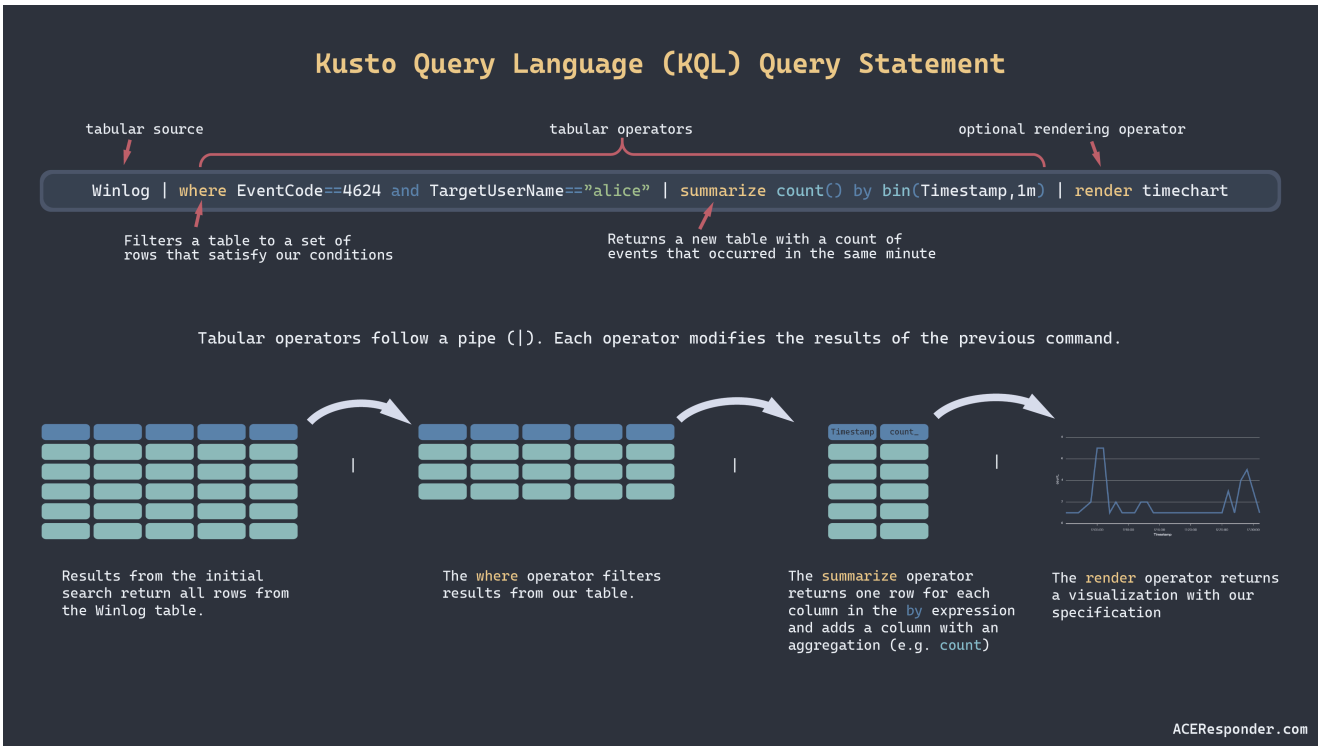
နောက်ထပ် အရေးကြီးတာက Threat Hunting နဲ့ Incident Triage အတွက် အသုံးဝင်တဲ့ Query နဲ့ Visualization တွေကို ဘယ်လို တည်ဆောက်ရမလဲ ဆိုတာကို သင်ပေးထားတာပါ။ Threat Hunting ဆိုတာက တိုက်ခိုက်မှု ဖြစ်ပြီးမှ စောင့်နေတာမဟုတ်ဘဲ Log Data ထဲမှာ သံသယဖြစ်စရာ လက္ခဏာတွေကို ကိုယ်တိုင် ရှာဖွေတာဖြစ်ပါတယ်။ Incident Triage ဆိုတာက ဖြစ်ပေါ်လာတဲ့ Security Incident တွေကို အရေးကြီးမှုအလိုက် ခွဲခြားပြီး ဘယ်ဟာကို အရင်ဆုံး ဖြေရှင်းရမလဲ ဆုံးဖြတ်တဲ့ လုပ်ငန်းစဉ် ဖြစ်ပါတယ်။ Visualization ဆိုတာက Data ကို ဇယား၊ Chart တွေအနေနဲ့ ပြသပေးတာဖြစ်ပြီး လူက မြန်မြန်နားလည်နိုင်အောင် ကူညီပေးပါတယ်။

How KQL Works

KQL ဘယ်လိုအလုပ်လုပ်သလဲဆိုတာကို ဆက်ပြီး ရှင်းပြရရင် KQL Query တစ်ခုကို အဆင့်ဆင့် တည်ဆောက်ရပါတယ်။ အဲဒီအဆင့်တွေကို Pipe လို့ခေါ်တဲ့ | ဆိုတဲ့ အမှတ်အသားနဲ့ ချိတ်ဆက်ထားပါတယ်။ ဒီ Pipe သဘောတရားက Linux Shell မှာ Command တွေကို | နဲ့ ချိတ်သုံးတဲ့ ပုံစံနဲ့ အရမ်းဆင်တူပါတယ်။ အဆင့်တစ်ခုချင်းစီက Data ကို ကိုယ်တိုင် ပြုပြင်ပြီး နောက်အဆင့်ကို ပို့ပေးပါတယ်။

ပထမဆုံး Data Source တစ်ခုကို ရွေးပါတယ်။ အဲဒီနောက် Filter လုပ်တာ၊ Transform လုပ်တာတွေကို တစ်ဆင့်ပြီး တစ်ဆင့် လုပ်သွားပါတယ်။ အဆုံးသတ်မှာ ကိုယ်လိုချင်တဲ့ Result ကို ရရှိလာတာ ဖြစ်ပါ

တယ်။ တစ်နည်းအားဖြင့် Kusto က Query ကို တစ်ကြိမ်တည်း ရေးတာမဟုတ်ဘဲ အလုပ်လုပ်စဉ်လို အဆင့်လိုက် စဉ်းစားပြီး တည်ဆောက်နိုင်အောင် လုပ်ပေးထားတာပါ။



ဥပမာအနေနဲ့ Windows Security Log တွေကို အရင်ယူပါတယ်။

```
Winlog
```

အဲဒီထဲက Logon Event ဖြစ်တဲ့ Event ID 4624 တွေကိုပဲ ရွေးထုတ်ပါတယ်။

```
| where EventCode == 4624
```

ထို့နောက် User Alice နဲ့ ပတ်သက်တဲ့ Logon Event တွေကိုသာ စစ်ဆေးပါတယ်။

```
and TargetName == "alice"
```

အဲဒီ Event တွေကို နာရီအလိုက် ရေတွက်ပြီး တစ်နာရီချင်းစီမှာ Logon ဘယ်လောက် ဖြစ်ခဲ့လဲ ဆိုတာကို တွက်ချက်ပါတယ်။ နောက်ဆုံးမှာ အဲဒီအချက်အလက်တွေကို Time Chart အနေနဲ့ ပြသပေးပါတယ်။

```
| summarize count() by bin(Timestamp,1m) | render time chart
```

ဒီလိုလုပ်ရင် Alice ရဲ့ Logon အပြုအမူက ပုံမှန်လား၊ ထူးခြားလား ဆိုတာကို မျက်စိနဲ့မြင်ရအောင် နားလည်နိုင်လာပါတယ်။

Exploring Kusto Data

Kusto ကို အသုံးပြုတဲ့အခါ Data တွေကို Database အတွင်းမှာ သိမ်းဆည်းထားပါတယ်။ ဒီ Kusto Database တစ်ခုအတွင်းမှာ Data တွေကို Table အနေနဲ့ ခွဲထားပြီး သိမ်းထားတာဖြစ်ပါတယ်။ Table ဆိုတာကို လွယ်လွယ်နားလည်ရရင် Excel Sheet တစ်ခုလိုပဲ ထင်နိုင်ပါတယ်။ Column တွေရှိပြီး Row တွေအနေနဲ့ Event Data တွေကို စုထားတာပါ။ Kusto နဲ့ အလုပ်လုပ်တဲ့အခါ အများအားဖြင့် Database တစ်ခုတည်းအတွင်းက Table တွေကိုပဲ အသုံးပြုပြီး Query တွေ ရေးကြပါတယ်။

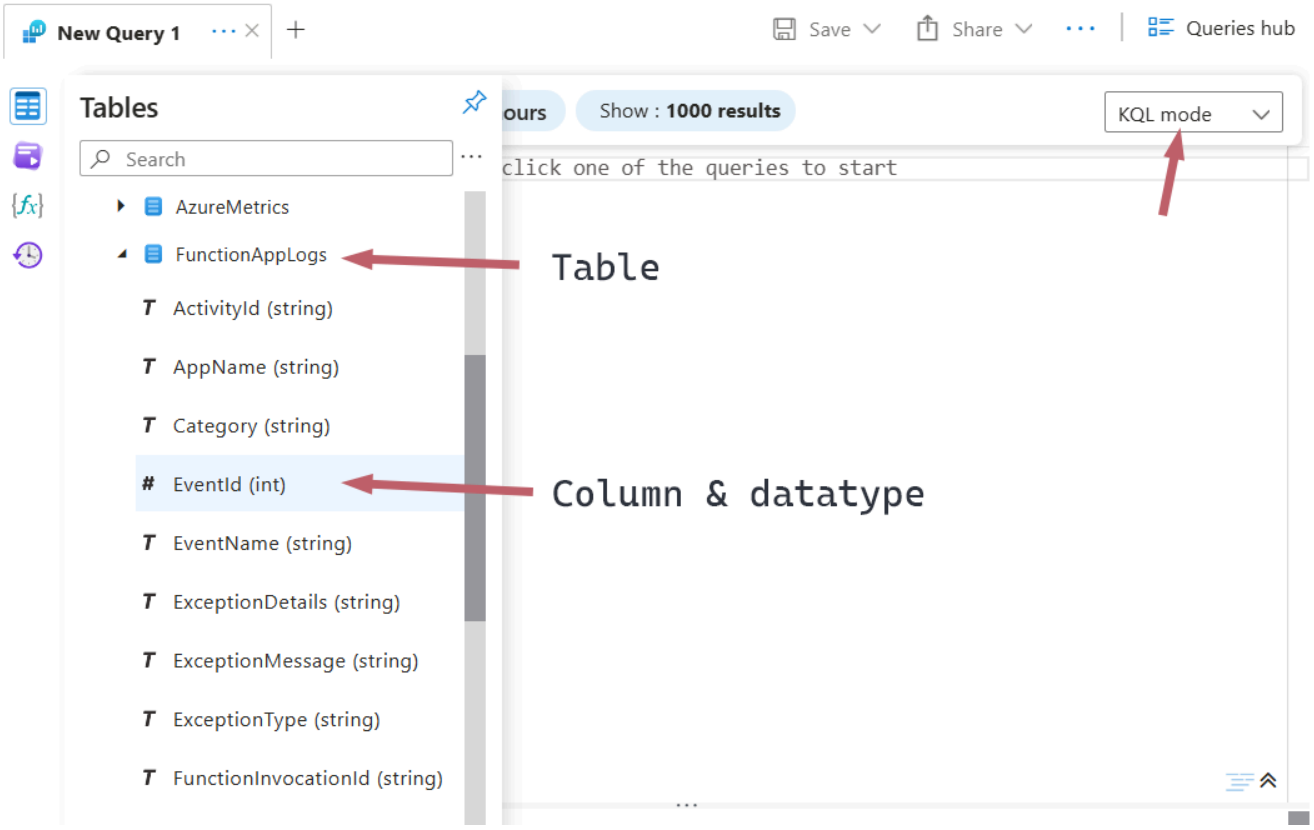
ဥပမာအားဖြင့် Log Analytics Workspace တစ်ခုမှာ အလုပ်လုပ်တဲ့အခါ Database ဆိုတဲ့ အယူအဆကို User က တိုက်ရိုက် မမြင်ရပါဘူး။ Database ကို Microsoft က နောက်ကွယ်မှာ စီမံထားပြီး User အနေနဲ့ ရရှိထားတဲ့ Table တွေကိုပဲ မြင်ရပြီး အဲဒီ Table တွေအတွင်းက Data တွေနဲ့ Query တွေ ရေးရတာ ဖြစ်ပါတယ်။ ဆိုလိုတာက Database ရဲ့ဖွဲ့စည်းပုံကို စိတ်ပူစရာမလိုဘဲ Table တွေကို အဓိကထားပြီး အလုပ်လုပ်ရတာပါ။

အကယ်၍ Azure Data Explorer လို Kusto Environment အပြည့်အစုံ ပါဝင်တဲ့ စနစ်တစ်ခုမှာ အလုပ်လုပ်နေတယ်ဆိုရင် Database ထဲမှာ ရှိတဲ့ Table စာရင်းကို ကိုယ်တိုင် Query နဲ့ ခေါ်ကြည့်နိုင်ပါတယ်။ အဲဒီလို Table စာရင်းကို ကြည့်ဖို့ Management Command လို့ခေါ်တဲ့ အထူး Command တွေကို အသုံးပြုရပါတယ်။ Native Table တွေကို ကြည့်ချင်ရင် `.show tables` ဆိုတဲ့ Command ကို သုံးနိုင်ပြီး External Source က ချိတ်ဆက်ထားတဲ့ Table တွေကို ကြည့်ချင်ရင် `.show external tables` ဆိုတဲ့ Command ကို သုံးပါတယ်။

ဒီ Management Command တွေရဲ့ ထူးခြားချက်က Dot(.) အမှတ်အသားနဲ့ စပြီး Query ရဲ့ အစမှာပဲ ထားရေးရတာ ဖြစ်ပါတယ်။ ဒီ Command တွေကို Data Query အတွက် မဟုတ်ဘဲ Database ကို စီမံခန့်ခွဲဖို့ အတွက် အသုံးပြုပါတယ်။ ဥပမာအားဖြင့် Table အသစ်တစ်ခု ဖန်တီးတာ၊ User တွေကို Permission ပေးတာ၊ Database Setting တွေ ပြင်ဆင်တာတွေကို ဒီ Management Command တွေနဲ့ လုပ်ပါတယ်။

ဒါပေမယ့် Kusto ကို အခြေခံထားတဲ့ Service အများစုမှာ User တွေကို Management Command အသုံးပြုခွင့် မပေးထားတာများပါတယ်။ အကြောင်းက Database စီမံခန့်ခွဲမှုတွေကို Service Provider က ကိုယ်စားလုပ်ပေးထားပြီး Security နဲ့ Stability အတွက် User ကို တိုက်ရိုက် ထိန်းချုပ်ခွင့် မပေးလိုတဲ့ အတွက် ဖြစ်ပါတယ်။ Log Analytics Workspace လို Service တွေမှာ Table တွေနဲ့ Schema ကို GUI ကနေ တိုက်ရိုက် ကြည့်ရှုနိုင်အောင် ပြုလုပ်ပေးထားပါတယ်။

SIEM Environment အတွင်းမှာလည်း အတူတူပါပဲ။ SIEM ထဲမှာ Management Command တွေကို တိုက်ရိုက် မဖွင့်ပေးထားပါဘူး။ ဒါပေမယ့် User အနေနဲ့ ဘယ် Table တွေ ရှိလဲ၊ Table တစ်ခုချင်းစီမှာ ဘယ် Column တွေ ပါလဲ ဆိုတာကို မသိရဘူးဆိုရင် Query ရေးလို့ မရပါဘူး။ အဲဒီအတွက် Table Name နဲ့ Schema တွေကို Query Window အပေါ်ဘက်မှာရှိတဲ့ အပြာရောင် Button တွေကနေ ပြသပေးထားပါတယ်။



အဲဒီအပြင် **IntelliSense** လို့ခေါ်တဲ့ အထောက်အကူပြု စနစ်ကိုလည်း ပေးထားပါတယ်။ IntelliSense ဆိုတာက Query ရေးနေတဲ့အချိန်မှာ Table Name တွေ၊ Column Name တွေ၊ Function တွေကို အလိုအလျောက် အကြံပြုပေးတဲ့ စနစ် ဖြစ်ပါတယ်။ User က Query ကို စာလုံးရိုက်လိုက်တာနဲ့ သက်ဆိုင်ရာ အကြံပြုချက်တွေ ပေါ်လာပြီး အမှားနည်းအောင် ကူညီပေးပါတယ်။ အလိုအလျောက် မပေါ်လာဘူးဆိုရင် `ctrl + space` သို့မဟုတ် `ctrl + i` ကို နှိပ်ပြီး လက်ဖြင့် ခေါ် နိုင်ပါတယ်။

External Tables

Kusto မှာ Data Table အမျိုးအစား နှစ်မျိုးရှိပါတယ်။ ပထမတစ်မျိုးက ပုံမှန် Table ဖြစ်ပြီး အဲဒီ Table ထဲက Data တွေကို Kusto Cluster ရဲ့ကိုယ်ပိုင် Storage အတွင်းမှာ တိုက်ရိုက် သိမ်းထားပါတယ်။ ဒုတိယတစ်မျိုးက External Table လို့ခေါ်တဲ့ Table ဖြစ်ပြီး ဒီ Table က Virtual Table လိုပဲ လုပ်ဆောင်ပါတယ်။ Virtual Table ဆိုတာက Table ပုံစံနဲ့ မြင်ရပေမယ့် Data ကို Kusto Cluster အတွင်းမှာ မသိမ်းဘဲ အပြင်ဘက် နေရာတစ်ခုမှာ သိမ်းထားတာကို ဆိုလိုပါတယ်။

External Table ထဲမှာပါတဲ့ Data တွေဟာ Kusto Cluster အပြင်ဘက်မှာ တည်ရှိနေပါတယ်။ ဥပမာ အားဖြင့် Azure Storage ထဲမှာ ဖြစ်နိုင်သလို SQL Database ထဲမှာလည်း ဖြစ်နိုင်ပါတယ်။ User က External Table ကို Query လုပ်တဲ့အခါ Kusto Cluster က အဲဒီ အပြင်ဘက်မှာရှိတဲ့ Data ကို ယာယီ ဆွဲယူလာပြီး သတ်မှတ်ထားတဲ့ Schema အတိုင်း ပြန်ညှိပေးပါတယ်။ အဲဒီနောက် Kusto Table ပုံစံအတိုင်း User ကို ပြသပေးတာဖြစ်ပါတယ်။

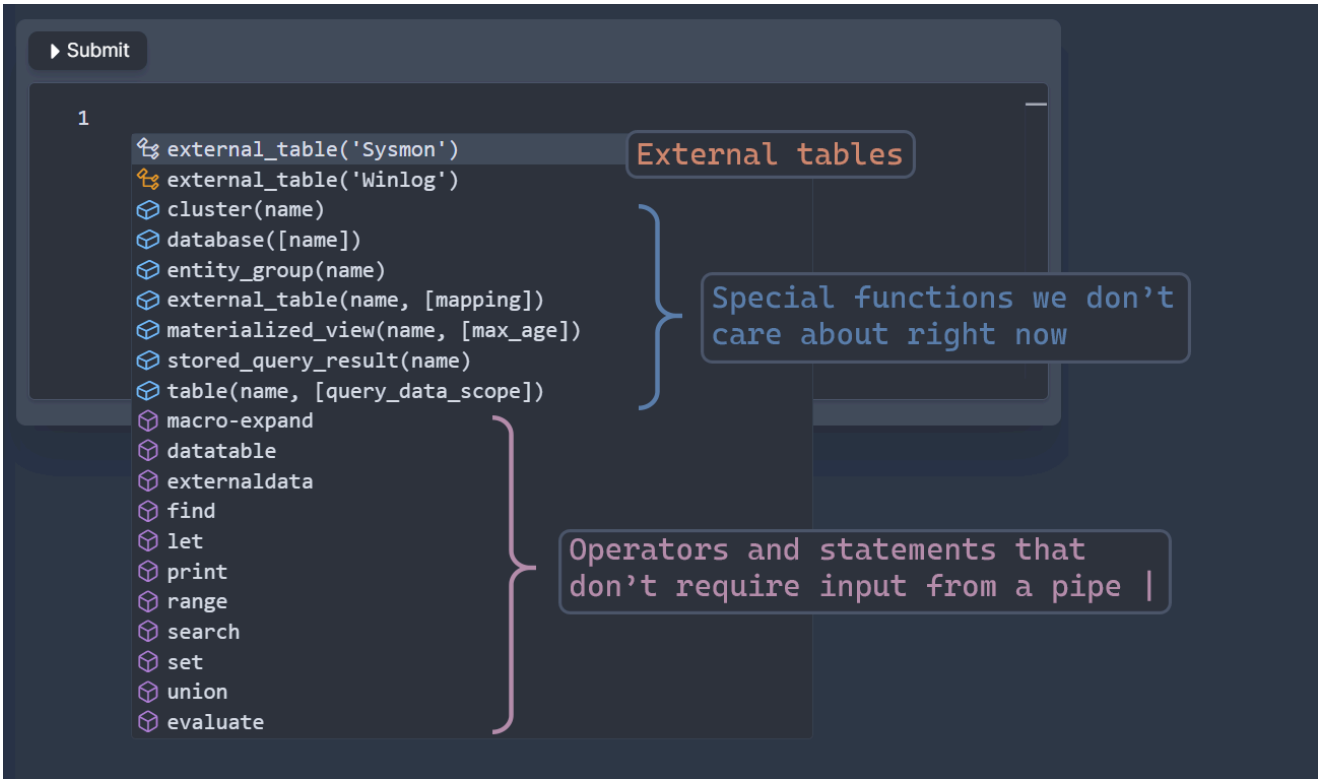
> ဒီနေရာမှာ အရေးကြီးဆုံး မှတ်ထားရမယ့် အချက်က External Table တွေဟာ Internal Table တွေနဲ့ အလုပ်လုပ်ပုံအရ တူတူပဲ ဖြစ်ပါတယ်။ Query ရေးတဲ့အခါ Internal Table လား External Table လားဆိုတာကို စိတ်ပူစရာ မလိုပါဘူး။ KQL Query အနေနဲ့ ကြည့်ရင် အားလုံးကို Table တစ်ခုလိုပဲ အသုံးပြုနိုင်ပါတယ်။ MS Sentinel သို့မဟုတ် Log Analytics လို SIEM Platform တွေမှာ အလုပ်

> လုပ်တဲ့အခါ External Table ကို တိုက်ရိုက် ခေါ်တဲ့ Function ကို မသုံးဘဲ Table Name ကိုပဲ သုံးရပါတယ်။

ဒီအကြောင်းအရာနဲ့ ဆက်စပ်ပြီး IntelliSense ရဲ့ အရေးပါမှုကိုလည်း နားလည်ထားဖို့ လိုပါတယ်။ IntelliSense ဆိုတာက Query Editor က User ရဲ့ Context ကို နားလည်ပြီး သင့်တော်တဲ့ အကြံပြုချက်တွေကို ပေးတဲ့ စနစ် ဖြစ်ပါတယ်။ User က Ctrl + Space ကို နှိပ်လိုက်တဲ့အခါ IntelliSense က လက်ရှိ အခြေအနေမှာ အသုံးပြုနိုင်မယ့် Option တွေကိုပဲ ပြသပေးပါတယ်။

ဥပမာအားဖြင့် Query Editor ကို အလွတ်ထားပြီး Ctrl + Space ကို နှိပ်လိုက်ရင် IntelliSense က Tabular Source အဖြစ် အသုံးပြုနိုင်တဲ့ Function တွေ၊ Operator တွေကို အရင်ဆုံး အကြံပြုပါတယ်။ ဘာကြောင့်လဲဆိုတော့ KQL Query တစ်ခုဟာ အမြဲတမ်း Data Source တစ်ခုနဲ့ စတင်ရပြီး အဲဒီ Source ပေါ်မှာ Operator တွေကို တစ်ဆင့်ပြီး တစ်ဆင့် ထပ်တိုးပြီး Data ကို ပြုပြင်သွားရတဲ့ ပုံစံဖြစ်လို့ပါ။

ဒီအခြေအနေမှာ IntelliSense က External Table နှစ်ခုဖြစ်တဲ့ Winlog နဲ့ Sysmon ကို ပြသပေးပါတယ်။ ဒါဟာ SIEM ထဲမှာ Windows Event Log နဲ့ Sysmon Event Log တွေကို ကိုယ်စားပြုတဲ့ Table တွေ ဖြစ်ပါတယ်။ အဲဒီအပြင် `cluster()` နဲ့ `database()` လို Function တွေကိုလည်း ပြသပေးပါတယ်။ အဲဒီ Function တွေကို သုံးရင် အခြား Kusto Cluster သို့မဟုတ် Database ထဲက Data ကို ချိတ်ဆက်ပြီး Query လုပ်နိုင်ပါတယ်။ ထို့အပြင် `union` နဲ့ `find` လို Operator နဲ့ Statement အချို့ကိုလည်း အကြံပြုပါတယ်။ အဲဒီ Operator တွေက Table အများကြီးကို ပေါင်းကြည့်တာ၊ Data ကို အပြည့်အဝ ရှာဖွေတာလို အလုပ်တွေကို လုပ်ပေးပါတယ်။



ဒီနေရာမှာ SOC Analyst တစ်ယောက်အနေနဲ့ အဓိက စိတ်ဝင်စားရမယ့် အရာက External Table တွေဖြစ်ပြီး အနည်းငယ်တော့ Operator နဲ့ Statement တွေလည်း လိုအပ်လာပါလိမ့်မယ်။ အဲဒီအထဲမှာ အရေးကြီးတဲ့ Statement တစ်ခုက `let` ဖြစ်ပါတယ်။

> `let` ဆိုတာက Query အတွင်းမှာ Expression တစ်ခုကို Variable အနေနဲ့ သိမ်းထားဖို့ အသုံးပြုပါတယ်။ ဒီသဘောတရားကို နောက်ပိုင်းမှာ ပိုပြီး အသေးစိတ် လေ့လာသွားရမှာ ဖြစ်ပါတယ်။

Table Structure

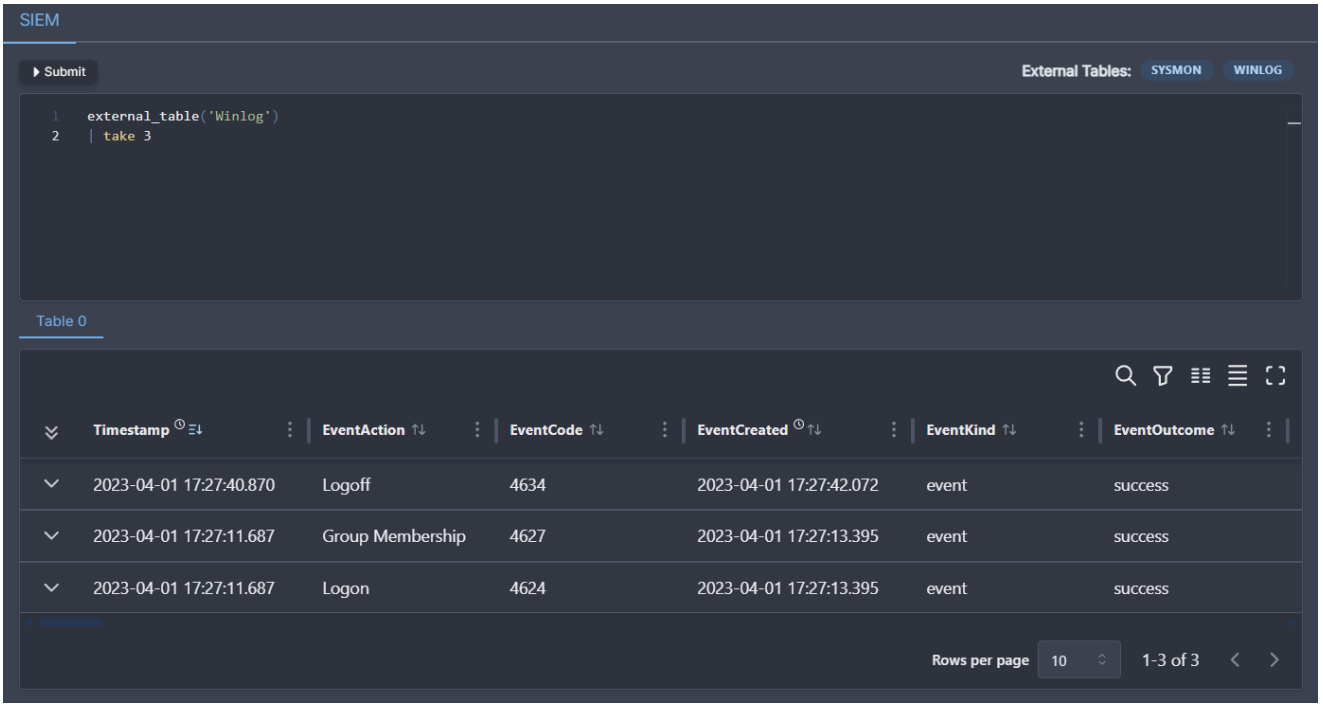
Kusto ထဲမှာရှိတဲ့ Table တစ်ခုချင်းစီဟာ Column နဲ့ Row တွေနဲ့ ဖွဲ့စည်းထားပါတယ်။ ဒီပုံစံက Database သို့မဟုတ် Excel Sheet တစ်ခုနဲ့ အလွန်ဆင်တူပါတယ်။ Row တစ်ခုချင်းစီက Event တစ်ခုစီကို ကိုယ်စားပြုပြီး Column တစ်ခုချင်းစီက အဲဒီ Event နဲ့ ပတ်သက်တဲ့ အချက်အလက်တစ်ခုစီကို ကိုယ်စားပြုပါတယ်။ ဥပမာအားဖြင့် `Timestamp` ဆိုတဲ့ Column က Event ဖြစ်ပေါ် ခဲ့တဲ့ အချိန်ကို သိမ်းထားပြီး `WorkstationName` ဆိုတဲ့ Column က Event ဖြစ်ပေါ် ခဲ့တဲ့ ကွန်ပျူတာအမည်ကို သိမ်းထားပါတယ်။

Kusto Table တစ်ခုမှာ Column တစ်ခုချင်းစီကို သတ်မှတ်ထားတဲ့ Data Type တစ်ခုစီ ရှိပါတယ်။ ဒီ Data Type ကို Scalar Data Type လို့ ခေါ် ပြီး string, datetime, int လို ပုံစံမျိုးတွေ ဖြစ်ပါတယ်။ Data Type တွေကို Table Schema ထဲမှာ ကြိုတင် သတ်မှတ်ထားပြီးသား ဖြစ်ပါတယ်။ Schema ဆိုတာက Table ရဲ့ဖွဲ့စည်းပုံကို သတ်မှတ်ထားတဲ့ စည်းမျဉ်းလိုပဲ နားလည်နိုင်ပါတယ်။

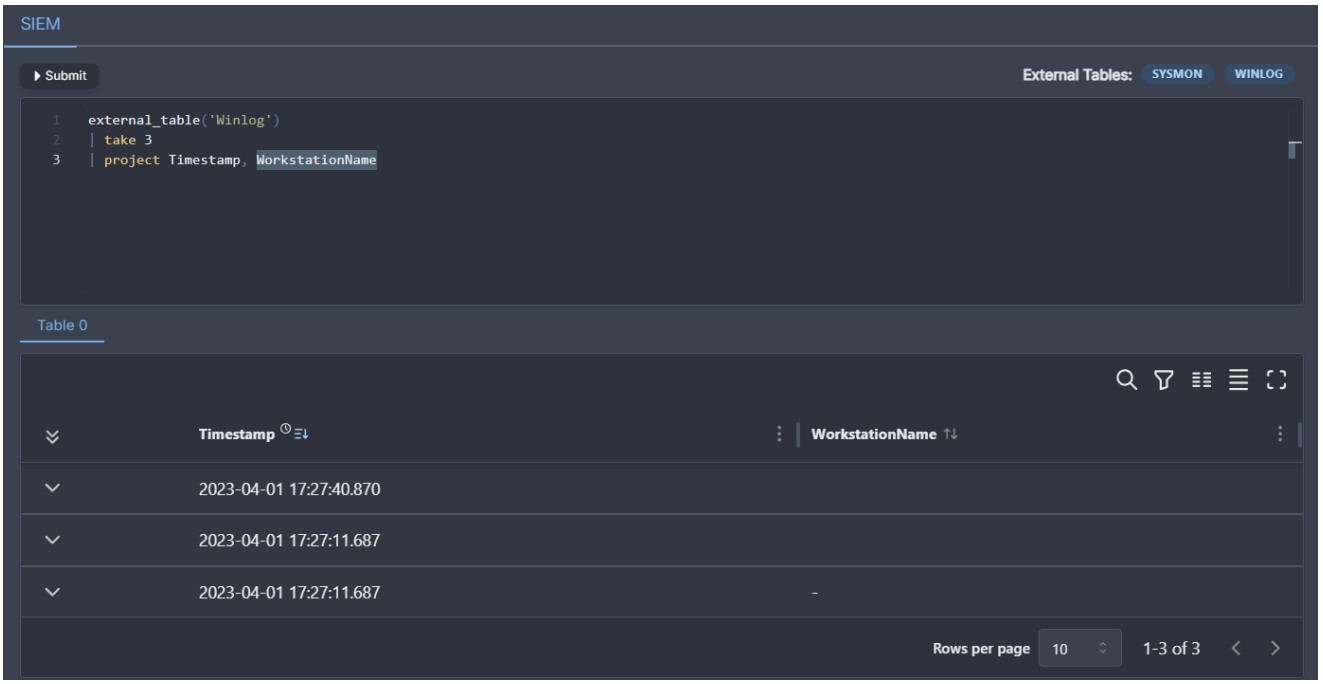
SIEM အတွင်းမှာ Table တစ်ခုရဲ့ Structure ကို ကြည့်ချင်ရင် ပထမဆုံး အဲဒီ Table ထဲက Data အနည်းငယ်ကို Query နဲ့ ခေါ် ကြည့်ရပါတယ်။ Query ကို Run လိုက်တဲ့အခါ Results Pane ထဲမှာ Data တွေ ပေါ်လာပြီး အပေါ်ဆုံးမှာ Column Header တွေကို မြင်ရပါမယ်။ အဲဒီ Column Header တွေက Column Name တွေ ဖြစ်ပါတယ်။ ဒီအဆင့်မှာ Column ဘယ်နှစ်ခု ရှိလဲ၊ Column အမည်တွေက ဘာတွေလဲ ဆိုတာကို နားလည်နိုင်ပါပြီ။

အကယ်၍ Column တစ်ခုရဲ့ Data Type ကို အတိအကျ သိချင်တယ်ဆိုရင် Column Name ပေါ်ကို Mouse ကို တင်ကြည့်ရင် Data Type ကို ပြပေးပါတယ်။ ဒီအချက်က အရေးကြီးပါတယ်။ ဘာကြောင့်လဲဆိုတော့ KQL Query ရေးတဲ့အခါ Data Type မတူရင် Operator နဲ့ Function တွေ အသုံးပြုလို့ မရတဲ့ အခြေအနေတွေ ဖြစ်တတ်လို့ပါ။

Query စမ်းရေးတဲ့အချိန်မှာ Result ကို အလွန်အကျွံ မထုတ်သင့်ပါဘူး။ Log Data တွေဟာ အရမ်းများတဲ့အတွက် Query တစ်ခါ Run လိုက်ရင် Row ထောင်နဲ့သောင်းနဲ့ ပြန်လာနိုင်ပါတယ်။ ဒါကြောင့် စမ်းသပ်တဲ့ Query တွေမှာ Result ကို ကန့်သတ်ထားတာက အလေ့အကျင့်ကောင်းတစ်ခု ဖြစ်ပါတယ်။ KQL မှာ အဲဒီလို ကန့်သတ်ဖို့ `take` ဆိုတဲ့ Operator ကို အသုံးပြုပါတယ်။ `take 10` လို့ ရေးလိုက်ရင် Result ထဲမှာ Row 10 ခုသာ ပြန်လာမှာ ဖြစ်ပါတယ်။ ဒီလိုလုပ်ခြင်းက Query ကို မြန်မြန် Run လို့ရစေပြီး မလိုအပ်တဲ့ Data တွေ မပြန်လာအောင် ကာကွယ်ပေးပါတယ်။



တစ်ချိန်တည်းမှာပဲ Column တွေကိုလည်း ကိုယ်လိုအပ်တဲ့ Column တွေချည်းပဲ ပြန်လာအောင် လုပ်နိုင်ပါတယ်။ ဒီအတွက် project ဆိုတဲ့ Operator ကို အသုံးပြုပါတယ်။ project ကို သုံးတဲ့အခါ Query Result ထဲမှာ ပြချင်တဲ့ Column Name တွေကို ကိုယ်တိုင် သတ်မှတ်ပေးရပါတယ်။ ဥပမာအားဖြင့် Timestamp နဲ့ WorkstationName ကိုပဲ ကြည့်ချင်တယ်ဆိုရင် project Timestamp, WorkstationName လို့ ရေးလိုက်ရုံနဲ့ အဲဒီ Column နှစ်ခုကိုပဲ Result ထဲမှာ ပြပေးပါလိမ့်မယ်။



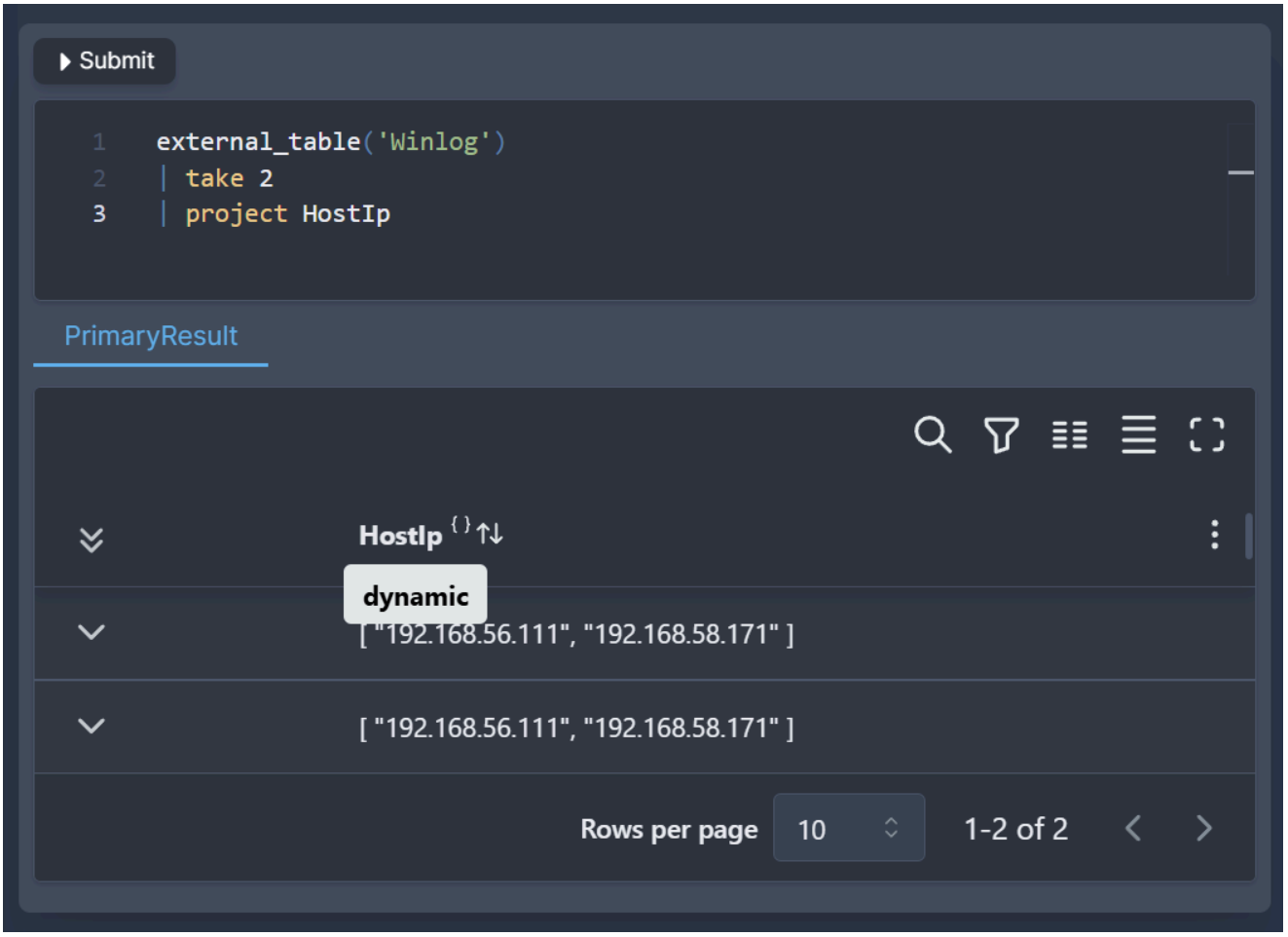
ဒီသဘောတရားတွေကို နားလည်ထားရင် SIEM ထဲမှာ KQL Query ရေးတဲ့အခါ Table Structure ကို မျက်စိနဲ့ မြင်သလို စဉ်းစားနိုင်လာပြီး လိုအပ်တဲ့ Data ကို မြန်မြန်နဲ့ တိတိကျကျ ဆွဲထုတ်နိုင်လာပါလိမ့်မယ်။

Test : Sampling Data

Use the `take` operators to take a small number of events from the `Winlog` external table. Use the `project` operator to return the `HostIp` field. What datatype is `HostIp`?

SIEM ထဲမှာ Log Data တွေကို စတင်လေ့လာတဲ့အချိန်မှာ Data အားလုံးကို တစ်ခါတည်း ကြည့်ဖို့ မသင့်တော်ပါဘူး။ Winlog လို Table တွေထဲမှာ Event Data က အလွန်အများကြီး ရှိနေတတ်ပြီး Query တစ်ခါ Run လိုက်ရင် Result တွေ များလွန်းသွားနိုင်ပါတယ်။ ဒါကြောင့် Data ကို နမူနာအနည်းငယ်သာ ယူပြီး စမ်းကြည့်တဲ့ အလေ့အကျင့်ကို Sampling Data လို့ ခေါ်ပါတယ်။ ဒီလို Sampling လုပ်ခြင်းက Table Structure ကို နားလည်ဖို့၊ Column တွေရဲ့ Data Type ကို စစ်ဆေးဖို့ အရမ်းအသုံးဝင်ပါတယ်။

ဒီအလုပ်ကို လုပ်ဖို့ KQL မှာ take operator ကို အသုံးပြုပါတယ်။ take ဆိုတာက Table ထဲက Event အနည်းငယ်ကိုပဲ ယူလာပေးတဲ့ Operator ဖြစ်ပါတယ်။ ဥပမာအားဖြင့် Winlog External Table ထဲက Event အနည်းငယ်ကိုပဲ ယူပြီး HostIp ဆိုတဲ့ Field ကိုပဲ ကြည့်ချင်တယ်ဆိုရင် take နဲ့ project ကို တွဲသုံးရပါတယ်။ ဒီလို Sampling Query ကို Run လိုက်တဲ့အခါ `HostIp` Column ကို မြင်ရပြီး အဲဒီ Column ရဲ့ Data Type ကိုလည်း စစ်ဆေးနိုင်ပါတယ်။



`HostIp` Column ရဲ့ Data Type ကို ကြည့်လိုက်ရင် dynamic ဆိုတာကို တွေ့ရပါလိမ့်မယ်။ dynamic ဆိုတဲ့ Data Type က Kusto မှာ အထူးအရေးကြီးပြီး အနည်းငယ် နားလည်ရခက်နိုင်ပါတယ်။ dynamic ဆိုတာက Data ပုံစံတစ်မျိုးတည်း မသတ်မှတ်ထားဘဲ အမျိုးမျိုး ဖြစ်နိုင်တယ်လို့ ဆိုလိုပါတယ်။ တစ်ခါတစ်ရံမှာ int သို့မဟုတ် string လို Scalar Value တစ်ခုသာ ဖြစ်နိုင်ပြီး တစ်ခါတစ်ရံမှာ Array ပုံစံနဲ့ Value အများကြီး ပါလာနိုင်ပါတယ်။ ဥပမာအားဖြင့် IP Address နှစ်ခု၊ သုံးခုကို Array အနေနဲ့ သိမ်းထားနိုင်ပါတယ်။ ထို့အပြင် Key နဲ့ Value တွေ ပါဝင်တဲ့ Object ပုံစံနဲ့လည်း ဖြစ်နိုင်ပါတယ်။

HostIp က dynamic ဖြစ်တဲ့အတွက် IP Address တစ်ခုတည်း မဟုတ်ဘဲ IP Address အများကြီး ပါလာနိုင်ပါတယ်။ အကယ်၍ HostIp ထဲက Value က Array ဖြစ်နေတယ်ဆိုရင် အဲဒီ Array ထဲက Element တစ်ခုချင်းစီကို Index နဲ့ ခေါ်ယူနိုင်ပါတယ်။ ဥပမာအားဖြင့် HostIp Array ထဲက ပထမဆုံး IP Address ကိုသာ ယူချင်ရင် အောက်ပါပုံစံနဲ့ ရေးနိုင်ပါတယ်။

```
external_table('Winlog')
| project HostIp[0]
```

ဒီ Query က HostIp Column ထဲမှာ Array ဖြစ်နေတဲ့ Value တွေအတွက် ပထမဆုံး IP Address ကိုပဲ ပြန်ပေးပါလိမ့်မယ်။ ဒီလို Array Access လုပ်နိုင်ခြင်းက Log Data ထဲမှာ Network Information တွေကို ခွဲခြမ်းစိတ်ဖြာတဲ့အခါ အရမ်းအသုံးဝင်ပါတယ်။

dynamic Field တစ်ခုက Object ပုံစံ ဖြစ်နေရင်လည်း အဲဒီ Object ထဲက Property တွေကို ခေါ်ယူနိုင်ပါတယ်။ Object ဆိုတာက key နဲ့ value တွေ ပါဝင်တဲ့ ဖွဲ့စည်းပုံဖြစ်ပြီး JSON နဲ့ ဆင်တူပါတယ်။ ဒီလို Object ထဲက Value တွေကို Dot Notation သို့မဟုတ် Bracket Notation နဲ့ ခေါ်ယူနိုင်ပါတယ်။ ဥပမာ

အားဖြင့် EventData ဆိုတဲ့ dynamic Field ထဲမှာ SubjectUserName နဲ့ TargetUserName ဆိုတဲ့ Property တွေ ပါနေရင် အောက်ပါပုံစံနဲ့ ခေါ် နိုင်ပါတယ်။

```
external_table('Winlog')
| project EventData.SubjectUserName, EventData["TargetUserName"]
```

ဒီ Query က EventData Object ထဲက SubjectUserName နဲ့ TargetUserName ကို ခွဲထုတ်ပြီး ပြပေးမှာ ဖြစ်ပါတယ်။ ဒါပေမယ့် ဒီဥပမာကို ဒီ Dataset မှာတော့ တိုက်ရိုက် အသုံးမချနိုင်ပါဘူး။ အကြောင်းက ဒီ SIEM Dataset မှာ EventData ထဲက Value တွေကို အရင်ကတည်းက Column အဖြစ် ခွဲထုတ်ပြီး SubjectUserName နဲ့ TargetUserName Column တွေအနေနဲ့ သိမ်းထားပြီးသား ဖြစ်လို့ပါ။

Filtering Data

SIEM ထဲမှာ Log Data တွေကို စုံစမ်းစစ်ဆေးတဲ့အခါ အရေးအကြီးဆုံး အလုပ်တစ်ခုက ကိုယ်လိုအပ်တဲ့ Event တွေကိုသာ ရွေးထုတ်ခြင်း ဖြစ်ပါတယ်။ Log Data တွေဟာ အလွန်များပြီး အရာအားလုံးကို ကြည့်နေရင် အရေးပါတဲ့ အချက်ကို လွယ်လွယ် မတွေ့နိုင်ပါဘူး။ ဒီလိုအချိန်မှာ KQL ရဲ့ where operator ကို အသုံးပြုပြီး Data ကို စစ်ထုတ်ရပါတယ်။

`where` operator ရဲ့ အလုပ်လုပ်ပုံကို နားလည်ရင် အရမ်းရိုးရှင်းပါတယ်။ Pipe ကနေ ဝင်လာတဲ့ Table Result ကို Row တစ်ခုချင်းစီ စစ်ဆေးပြီး ကိုယ်သတ်မှတ်ထားတဲ့ အခြေအနေကို true ဖြစ်တဲ့ Row တွေကိုပဲ ပြန်ပေးပါတယ်။ Condition ကို မကိုက်ညီတဲ့ Row တွေကိုတော့ ဖယ်ရှားလိုက်ပါတယ်။ ဒါကြောင့် where ဆိုတာ “စစ်ထုတ်တဲ့ စစ်ကာ” လိုပဲ ထင်နိုင်ပါတယ်။

where operator ထဲမှာ Condition ကို ရေးတဲ့အခါ နည်းလမ်းအမျိုးမျိုး အသုံးပြုနိုင်ပါတယ်။ Query ရေးတဲ့အခါ အရေးကြီးတဲ့ အလေ့အကျင့်က where operator ကို Data Source နောက်မှာ ချက်ချင်း သုံးတာပါ။

> Table ကို ခေါ် ပြီးသားနောက်မှာ where နဲ့ စစ်ထုတ်လိုက်ရင် နောက်ထပ် Operator တွေက Data နည်းနည်းပဲ ကိုင်တွယ်ရတော့တဲ့အတွက် Performance ပိုကောင်းပြီး Resource သုံးစွဲမှုလည်း လျော့နည်းပါတယ်။ ဒါဟာ SOC Analyst တစ်ယောက်အတွက် အလွန်အရေးကြီးတဲ့ Best Practice ဖြစ်ပါတယ်။

Logical Operator

Logical Operator ဆိုတာက Scalar Value နှစ်ခုကို နှိုင်းယှဉ်ပြီး true သို့မဟုတ် false ကို ဆုံးဖြတ်ပေးတဲ့ Operator တွေ ဖြစ်ပါတယ်။

Syntax	Description
`==`	Returns `true` if the operands are equal to each other.
`!=`	Returns `true` if the operands aren't equal to each other.
`and`	Returns `true` if both operands are `true`.
`or`	Returns `true` if either operand is `true`.

ဒီနေရာမှာ အရေးကြီးဆုံး နားလည်ထားရမယ့် အချက်က `and` operator ဟာ `or` operator ထက် ဦးစားပေးအဆင့် ပိုမြင့်ပါတယ်။ ဆိုလိုတာက Parentheses မသုံးဘဲ Condition တွေ ရေးလိုက်ရင် KQL က and ကို အရင်ဆုံး စစ်ပါတယ်။ ဒီအချက်ကို မသိဘဲ Query ရေးမိရင် Result က ကိုယ်မျှော်လင့်ထားတာနဲ့ လုံးဝ မတူသွားနိုင်ပါတယ်။

ဥပမာအားဖြင့် Alice နဲ့ Bob နှစ်ယောက်လုံးရဲ့ Logon Event ဖြစ်တဲ့ EventCode 4624 ကို ရှာချင်တယ်ဆိုပါစို့။ ဒီလိုအခြေအနေမှာ Query ကို အောက်ပါအတိုင်း ရေးရပါမယ်။

```
external_table('Winlog')
| where (TargetUserName=="alice" or TargetUserName=="bob") and
EventCode==4624
| take 10
```

ဒီ Query မှာ Parentheses ကို သုံးထားတာကြောင့် Alice ဖြစ်ဖြစ် Bob ဖြစ်ဖြစ် Logon Event ဖြစ်တဲ့ 4624 ကိုပဲ ပြန်ပေးပါလိမ့်မယ်။ ဒါဟာ SOC Analyst တစ်ယောက်အနေနဲ့ တကယ့်လိုအပ်တဲ့ Result ဖြစ်ပါတယ်။

အကယ်၍ Parentheses ကို မသုံးဘဲ Query ကို ရေးလိုက်မယ်ဆိုရင် KQL က and ကို အရင်ဆုံး စစ်သွားပါလိမ့်မယ်။ အဲဒီအခါ Alice အတွက် EventCode မသတ်မှတ်ဘဲ Event အားလုံး ပြန်လာပြီး Bob အတွက်သာ 4624 Event ကို ပြန်လာတဲ့ အခြေအနေ ဖြစ်သွားနိုင်ပါတယ်။ ဒီလို အမှားသေးသေးလေးတစ်ခုက Investigation Result ကို လုံးဝ မှားသွားစေနိုင်ပြီး Threat Detection မှာ ကြီးမားတဲ့ အကျိုးဆက်တွေ ဖြစ်စေနိုင်ပါတယ်။

String Operators

SIEM ထဲမှာ Log Data တွေကို စစ်ဆေးတဲ့အခါ စာသားအမျိုးအစား Data ကို အခြေခံပြီး Filter လုပ်ရတဲ့ အခြေအနေတွေ အများကြီး ရှိပါတယ်။ ဥပမာအားဖြင့် Username, Process Name, Command Line, Computer Name, File Path လို အချက်အလက်တွေဟာ အားလုံး string Data ဖြစ်ပါတယ်။ ဒီလို Data တွေကို တိတိကျကျ စစ်ထုတ်နိုင်ဖို့ KQL မှာ String Operator တွေကို အသုံးပြုရပါတယ်။

String Operator တွေထဲမှာ အခြေခံဆုံးက == နဲ့ != ဖြစ်ပါတယ်။ == ကို သုံးရင် ဘယ်ဘက်နဲ့ ညာဘက် Value တူညီလားဆိုတာ စစ်ပါတယ်။ != ကို သုံးရင် မတူညီဘူးလားဆိုတာ စစ်ပါတယ်။ ဒီ Operator နှစ်ခုဟာ Case-Sensitive ဖြစ်ပါတယ်။ ဆိုလိုတာက "Alice" နဲ့ "alice" ကို မတူဘူးလို့ သတ်မှတ်ပါတယ်။ Case ကို မစဉ်းစားချင်ဘူးဆိုရင် =~ နဲ့ !~ ကို အသုံးပြုရပါတယ်။ =~ က Case-Insensitive Equality ဖြစ်ပြီး !~ က Case-Insensitive Not Equal ဖြစ်ပါတယ်။

Operator	Description	Case-Sensitive
`contains`	Left contains the right. `"ACEResponder" contains "cer"`	
`endswith`	Left ends with right. `"ACEResponder" endswith "der"`	
`has`	Left contains the whole token on right. `"Kusto Tutorial" has "kusto"`	

Operator	Description	Case-Sensitive
`hasprefix`	Left has a token that begins with right. `"Kusto Tutorial" hasprefix "tuto"`	
`hassuffix`	Left has a token that ends with right. `"Kusto Tutorial" hassuffix "rial"`	
`in`	Left in the list on right. `"ACE" in ("ACE", "Responder")`	✓
`matches regex`	Left contains a match for regex on right. `"ACEResponder" matches regex "CE.*r"	✓
`startswith`	Left starts with right `"ACEResponder" startswith "ace"`	

Equality Operator တွေထက် ပိုပြီး အသုံးများတာက String Matching Operator တွေ ဖြစ်ပါတယ်။ `contains` ဆိုတဲ့ Operator က ဘယ်ဘက် string ထဲမှာ ညာဘက် string ပါဝင်လားဆိုတာ စစ်ပါတယ်။ ဥပမာအားဖြင့် "ACEResponder" ဆိုတဲ့ string ထဲမှာ "cer" ပါဝင်လားဆိုတာကို စစ်တာပါ။ ဒီ Operator ကို Process Name, Command Line လို စာကြောင်းရှည်တွေထဲမှာ သတ်မှတ်ထားတဲ့ စာသား ပါဝင်လားဆိုတာ စစ်ဖို့ မကြာခဏ အသုံးပြုပါတယ်။

`endswith` ဆိုတဲ့ Operator က string တစ်ခုရဲ့ အဆုံးပိုင်းမှာ သတ်မှတ်ထားတဲ့ စာသားနဲ့ ပြီးဆုံးလားဆိုတာ စစ်ပါတယ်။ ဥပမာအားဖြင့် "ACEResponder" ဆိုတဲ့ string က "der" နဲ့ ပြီးဆုံးလားဆိုတာကို စစ်တဲ့ ပုံစံပါ။ ဒါဟာ File Extension စစ်တဲ့အခါ အသုံးများပါတယ်။

`has` ဆိုတဲ့ Operator က contains နဲ့ ဆင်တူပေမယ့် Token အပြည့်အစုံကို စစ်ပါတယ်။ Token ဆိုတာက စကားလုံးအလိုက် ခွဲထားတဲ့ အပိုင်းကို ဆိုလိုပါတယ်။ ဥပမာအားဖြင့် "Kusto Tutorial" ဆိုတဲ့ string မှာ "kusto" ဆိုတဲ့ စကားလုံးတစ်လုံးလုံး ပါဝင်လားဆိုတာကို စစ်ပါတယ်။ ဒါကြောင့် has ဟာ contains ထက် ပိုတိကျပါတယ်။

`hasprefix` ဆိုတဲ့ Operator က Token တစ်ခုရဲ့ အစမှာ သတ်မှတ်ထားတဲ့ စာသားနဲ့ စတင်လားဆိုတာ စစ်ပါတယ်။ "Kusto Tutorial" ဆိုတဲ့ string ထဲမှာ "tuto" နဲ့ စတင်တဲ့ Token ရှိလားဆိုတာကို စစ်တာပါ။ အဲဒီမှာ "Tutorial" ဆိုတဲ့ Token က "tuto" နဲ့ စတင်တာကြောင့် true ဖြစ်ပါတယ်။

`hassuffix` ဆိုတဲ့ Operator က Token တစ်ခုရဲ့ အဆုံးမှာ သတ်မှတ်ထားတဲ့ စာသားနဲ့ ပြီးဆုံးလားဆိုတာ စစ်ပါတယ်။ "Kusto Tutorial" ဆိုတဲ့ string ထဲမှာ "rial" နဲ့ ပြီးဆုံးတဲ့ Token ရှိလားဆိုတာကို စစ်တဲ့ ပုံစံပါ။ ဒါဟာ Filename သို့မဟုတ် Domain Name စစ်တဲ့အခါ အသုံးဝင်ပါတယ်။

`in` ဆိုတဲ့ Operator က ဘယ်ဘက် Value က ညာဘက်မှာ ပေးထားတဲ့ စာရင်းထဲက တစ်ခုခုနဲ့ ကိုက်ညီလားဆိုတာ စစ်ပါတယ်။ ဥပမာအားဖြင့် "ACE" ဆိုတဲ့ string က ("ACE", "Responder") ဆိုတဲ့ စာရင်းထဲမှာ ပါဝင်လားဆိုတာကို စစ်တာပါ။ ဒီ Operator က Case-Insensitive ဖြစ်ပြီး Value အများကြီးနဲ့ နှိုင်းယှဉ်ရတဲ့အခါ အရမ်းအသုံးဝင်ပါတယ်။

`matches rege`x ဆိုတဲ့ Operator က Regular Expression ကို အသုံးပြုပြီး string ကို စစ်တာပါ။ ဥပမာအားဖြင့် "ACEResponder" ဆိုတဲ့ string က "CE.*r" ဆိုတဲ့ Regex Pattern နဲ့ ကိုက်ညီလားဆိုတာကို စစ်ပါတယ်။ Regex က အလွန်အစွမ်းထက်ပေမယ့် Performance ကို ထိခိုက်နိုင်တဲ့အတွက် လိုအပ်တဲ့အခါမှသာ အသုံးပြုသင့်ပါတယ်။ ဒီ Operator က Case-Insensitive ဖြစ်ပါတယ်။

`startswith` ဆိုတဲ့ Operator က string တစ်ခုရဲ့ အစမှာ သတ်မှတ်ထားတဲ့ စာသားနဲ့ စတင်လားဆိုတာ စစ်ပါတယ်။ ဥပမာအားဖြင့် "ACEResponder" ဆိုတဲ့ string က "ace" နဲ့ စတင်လားဆိုတာကို စစ်တဲ့ ပုံစံပါ။ Command Line နဲ့ Process Name စစ်တဲ့အခါ အလွန်အသုံးများပါတယ်။

ဒီ String Operator တွေအားလုံးမှာ Case Sensitivity ကို ထိန်းချုပ်နိုင်တဲ့ Suffix တွေ ရှိပါတယ်။ Operator က မူလအနေဖြင့် Case-Sensitive ဖြစ်ရင် Case-Insensitive ဖြစ်အောင် ~ ကို နောက်မှာ ထည့်ရပါတယ်။ ဥပမာအားဖြင့် in~ လိုပဲ သုံးနိုင်ပါတယ်။ တစ်ဖက်မှာ Operator က မူလအနေဖြင့် Case-Insensitive ဖြစ်နေတယ်ဆိုရင် Case-Sensitive ဖြစ်အောင် _cs ဆိုတဲ့ Suffix ကို သုံးရပါတယ်။ ဥပမာအားဖြင့် contains_cs လို့ သုံးရင် Case ကို တိတိကျကျ စစ်ပါလိမ့်မယ်။

The In and Between Operators

SIEM ထဲမှာ Log Data တွေကို စစ်ဆေးတဲ့အခါ Column တစ်ခုက Value တစ်ခုတည်းနဲ့ ကိုက်ညီလားဆိုတာထက် Value အများကြီးထဲက တစ်ခုခုနဲ့ ကိုက်ညီလား၊ သတ်မှတ်ထားတဲ့ အကွာအဝေးအတွင်း ပါဝင်လားဆိုတာကို စစ်ရတဲ့ အခြေအနေတွေ မကြာခဏ ကြုံရပါတယ်။ ဒီလိုအခြေအနေတွေကို ရိုးရိုး `or` operator တွေနဲ့ ရေးနိုင်ပေမယ့် KQL မှာ ပိုပြီး ဖတ်လွယ်၊ ပြင်လွယ်၊ အမှားနည်းအောင် ကူညီပေးတဲ့ `in` operator နဲ့ `between` operator ကို အသုံးပြုနိုင်ပါတယ်။

in operator

ပထမဦးဆုံး `in` operator ကို ကြည့်ရအောင်။ `in` operator ရဲ့ အဓိက ရည်ရွယ်ချက်က Column တစ်ခုရဲ့ Value ဟာ ပေးထားတဲ့ စာရင်းထဲက တစ်ခုခုနဲ့ ကိုက်ညီလားဆိုတာကို စစ်ဆေးခြင်း ဖြစ်ပါတယ်။ ယခင်က Alice သို့မဟုတ် Bob လားဆိုတာကို or operator နဲ့ စစ်ခဲ့ရပေမယ့် User အများကြီး ပါလာတဲ့အခါ or ကို ထပ်ခါထပ်ခါ ရေးရတာ အဆင်မပြေတော့ပါဘူး။ ဒီလိုအချိန်မှာ in operator ကို သုံးရင် Query ပိုပြီး ရှင်းလင်းလာပါတယ်။

ဥပမာအားဖြင့် Logon Event ဖြစ်တဲ့ EventCode 4624 ကို DC account ဖြစ်တဲ့ dc$ နဲ့ bob အတွက် ရှာချင်တယ်ဆိုပါစို့။ ဒီအခါ Query ကို အောက်ပါပုံစံနဲ့ ရေးနိုင်ပါတယ်။

```
external_table('Winlog')
| where TargetUserName in~ ("dc$", "bob") and EventCode==4624
| project EventCode, TargetUserName
```

ဒီ Query မှာ `in~` ကို သုံးထားတာကြောင့် Username ကို Case-Insensitive နဲ့ စစ်ပါတယ်။ ဆိုလိုတာက Bob, bob, BOB ဆိုပြီး ဘယ်လိုရေးထားပါစေ ကိုက်ညီရင် Result ထဲကို ပြန်ပေးပါလိမ့်မယ်။ ဒီလိုရေးခြင်းက Query ကို ပိုဖတ်လွယ်စေပြီး User စာရင်း ပြောင်းလဲရင်လည်း လွယ်လွယ် ပြင်နိုင်ပါတယ်။

`in` operator ကို string အတွက်ပဲ မဟုတ်ဘဲ scalar datatype အားလုံးနီးပါးမှာ အသုံးပြုနိုင်ပါတယ်။ ဥပမာအားဖြင့် Windows Logon Success ဖြစ်တဲ့ 4624 နဲ့ Logon Failure ဖြစ်တဲ့ 4625 Event နှစ်မျိုးလုံးကို တစ်ခါတည်း ဆွဲထုတ်ချင်တယ်ဆိုရင် in operator က အရမ်းအသုံးဝင်ပါတယ်။

```
external_table('Winlog')
| where EventCode in (4624, 4625)
```

```
| project EventCode, TargetUserName
```

ဒီ Query က Logon Success နဲ့ Failure နှစ်မျိုးလုံးကို ပြန်ပေးပြီး Brute Force Attack လို Threat တွေကို စတင် သုံးသပ်တဲ့အခါ အခြေခံအနေနဲ့ အသုံးချနိုင်ပါတယ်။ ဒီလိုအခြေအနေမှာ or operator နှစ်ကြိမ် ရေးမယ့်အစား in ကို သုံးတာက ပိုပြီး စနစ်ကျပါတယ်။

between operator

အခု `between` operator ကို ဆက်ကြည့်ရအောင်။ between operator ရဲ့ အလုပ်လုပ်ပုံက Value တစ်ခု ဟာ သတ်မှတ်ထားတဲ့ Range အတွင်း ပါဝင်လားဆိုတာကို စစ်ခြင်း ဖြစ်ပါတယ်။ ဒီ Operator က Range ကို ဘယ်ဘက်နဲ့ ညာဘက် နှစ်ဖက်စလုံး ပါဝင်တဲ့ Inclusive Range အနေနဲ့ စစ်ပါတယ်။

ဥပမာအားဖြင့် Network Investigation လုပ်တဲ့အခါ Destination Port က Dynamic Port လားဆိုတာ စစ်ရတဲ့အခြေအနေတွေ ရှိပါတယ်။ Windows မှာ Dynamic Port တွေဟာ ပုံမှန်အားဖြင့် 49152 ကနေ 65535 အထိ ဖြစ်ပါတယ်။ ဒီလို Port တွေကို အသုံးပြုတဲ့ Traffic က RPC Server နဲ့ ဆက်သွယ်နေတဲ့ Process ဖြစ်နိုင်တဲ့အတွက် သံသယထားစရာ ဖြစ်တတ်ပါတယ်။

```
external_table('Sysmon')
| where DestinationPort between (49152 .. 65535)
| project SourceIp, Image, DestinationIp, DestinationPort
```

ဒီ Query က DestinationPort ဟာ 49152 နဲ့ 65535 ကြားမှာ ပါဝင်တဲ့ Event တွေကိုပဲ ပြန်ပေးပါတယ်။ ဒီလို Filter လုပ်ပြီး SourceIp နဲ့ Image ကို ကြည့်ရင် ဘယ် Process က Dynamic Port ကို သုံးပြီး ဆက်သွယ်နေလဲဆိုတာကို ခွဲခြမ်းစိတ်ဖြာနိုင်ပါတယ်။

between operator ကို Time-based Investigation မှာလည်း အလွန်အရေးကြီးစွာ အသုံးပြုပါတယ်။ Incident တစ်ခု ဖြစ်ပွားခဲ့တဲ့ အချိန်မတိုင်ခင် မိနစ်အနည်းငယ်အတွင်း ဘာတွေ ဖြစ်ခဲ့လဲဆိုတာကို ပြန်လည် ကြည့်ရတဲ့အခါ Time Window ကို သုံးရပါတယ်။

```
let ts = datetime("2023-04-01T17:31:13.703Z");
external_table('Winlog')
| where Timestamp between (ts - 5m .. ts) and EventCode==4624
| project Timestamp, HostName
```

ဒီ Query ကို အဆင့်လိုက် နားလည်ရအောင် ရှင်းပြရရင် ပထမဆုံး သတ်မှတ်ထားတဲ့ Timestamp ကို datetime အဖြစ် ပြောင်းပြီး `ts` ဆိုတဲ့ Variable ထဲမှာ သိမ်းထားပါတယ်။ ဒုတိယအဆင့်မှာ Winlog Table ကို ခေါ်ပါတယ်။ ထို့နောက် where operator ကို သုံးပြီး `Timestamp` ဟာ `ts` မတိုင်ခင် ၅ မိနစ်က နေ ts အချိန်အထိ အတွင်းမှာ ပါဝင်တဲ့ Event တွေကို စစ်ထုတ်ပါတယ်။ အဲဒီ Event တွေထဲက Logon Event ဖြစ်တဲ့ 4624 ကိုပဲ ရွေးပြီး `HostName` နဲ့ အချိန်ကို ပြန်ပေးပါတယ်။

ဒီလို Query ရေးတဲ့အခါ အလွန်အရေးကြီးတဲ့ Best Practice တစ်ခုက Time Filter ကို အရင်ဆုံး သုံးတာပါ။ Kusto က Time Column တွေကို Index လုပ်ထားတဲ့အတွက် Time-based where condition ကို အရင်သုံးရင် Query က ပိုမြန်ပြီး Resource သုံးစွဲမှုလည်း ပိုထိရောက်ပါတယ်။ SOC Analyst တစ်ယောက် အနေနဲ့ ဒီအလေ့အကျင့်ကို မဖြစ်မနေ ယူထားသင့်ပါတယ်။

Aggregation

SIEM ထဲမှာ Log Data တွေကို စုံစမ်းစစ်ဆေးတဲ့အခါ Event တစ်ခုချင်းစီကို ကြည့်နေရုံနဲ့ မလုံလောက်ပါဘူး။ Event အများကြီးကို စုစည်းပြီး ပုံစံတစ်ခုအဖြစ် မြင်နိုင်ဖို့ လိုအပ်ပါတယ်။ ဒီလို Data အများကြီးကို တစ်ခုတည်းသော အချက်အလက်အဖြစ် စုစည်းတာကို Aggregation လို့ ခေါ်ပါတယ်။ ယခုအချိန်ထိ ကျွန်တော်တို့ လုပ်လာခဲ့တာက သတ်မှတ်ထားတဲ့ အခြေအနေကို ကိုက်ညီတဲ့ Event အရေအတွက်ကို ရေတွက်တာ ဖြစ်ပါတယ်။ ဒါဟာ Aggregation ရဲ့ အခြေခံဆုံး ပုံစံပါပဲ။

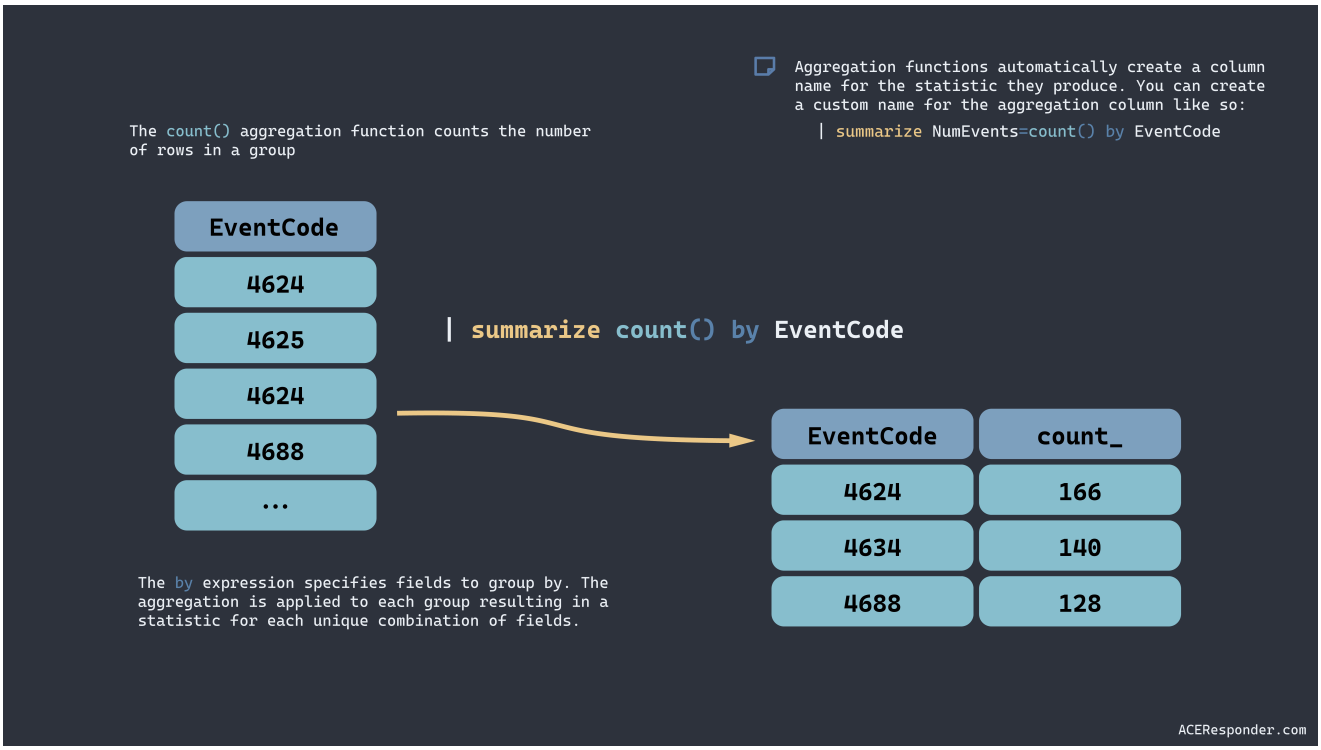
Aggregation က အရမ်း စိတ်ဝင်စားဖို့ ကောင်းလာတဲ့အချိန်က Data ကို Group အလိုက် ခွဲပြီး စုစည်းတဲ့အခါ ဖြစ်ပါတယ်။ ဥပမာအားဖြင့် Database ထဲမှာ ဘယ် EventCode က ဘယ်လောက် အကြိမ် ဖြစ်နေတယ်ဆိုတာကို သိချင်တယ်ဆိုရင် EventCode တစ်ခုချင်းစီကို Group လုပ်ပြီး Count လုပ်ရပါမယ်။ အဲဒီအခါ Result Table ထဲမှာ EventCode တစ်ခုနဲ့ အဲဒီ EventCode ဖြစ်တဲ့ အရေအတွက် ဆိုပြီး Column နှစ်ခု ပါလာပါလိမ့်မယ်။ ဒီလို Result က SOC Analyst တစ်ယောက်ကို စနစ်ထဲမှာ ဘာတွေ အများဆုံး ဖြစ်နေတယ်ဆိုတာကို တစ်ချက်ကြည့်နဲ့ နားလည်စေပါတယ်။

KQL မှာ Aggregation လုပ်ဖို့ `summarize` operator ကို အသုံးပြုပါတယ်။ `summarize` ရဲ့ အလုပ်လုပ်ပုံက အရမ်းရှင်းပါတယ်။ `by` ဆိုတဲ့ Keyword နောက်မှာ ပေးထားတဲ့ Column Value တူတဲ့ Row တွေကို Group လုပ်ပြီး အဲဒီ Group တစ်ခုချင်းစီအပေါ် Aggregation Function ကို အသုံးပြုပါတယ်။ ဥပမာအားဖြင့် EventCode အလိုက် Count လုပ်ချင်ရင် EventCode တူတဲ့ Row တွေကို တစ်စုတည်း စုစည်းပြီး အရေအတွက်ကို ရေတွက်ပေးပါတယ်။

```
external_table('Winlog')
| summarize count() by EventCode
| sort by count_ desc
```

ဒီ Query မှာ count() ဆိုတဲ့ Aggregation Function ကို အသုံးပြုထားပါတယ်။ summarize က EventCode တူတဲ့ Event တွေကို Group လုပ်ပြီး Group တစ်ခုချင်းစီမှာ Event ဘယ်လောက် ရှိလဲဆိုတာကို ရေတွက်ပါတယ်။ Result ထဲမှာ EventCode Column နဲ့ Kusto က အလိုအလျောက် ဖန်တီးပေးတဲ့ count_ Column ပါလာပါလိမ့်မယ်။ နောက်ဆုံးမှာ sort ကို သုံးပြီး Event အများဆုံး ဖြစ်တဲ့ EventCode ကို အပေါ်ဆုံးမှာ ပြထားပါတယ်။

ဒီလို Aggregation ကို အသုံးပြုခြင်းက Noise အများကြီးပါတဲ့ Log Data ထဲကနေ အရေးပါတဲ့ Pattern တွေကို မြင်နိုင်စေပါတယ်။ ဥပမာအားဖြင့် Logon Event တွေ အရမ်းများနေရင် သဘာဝလား၊ ထူးခြားလားဆိုတာကို EventCode အလိုက် Count ကြည့်ရုံနဲ့ စတင် သုံးသပ်နိုင်ပါတယ်။



count() နဲ့ ဆင်တူတဲ့ နောက်ထပ် Aggregation Function တစ်ခုက `dcount()` ဖြစ်ပါတယ်။ `dcount()` က Row အရေအတွက်ကို မရေတွက်ဘဲ Column တစ်ခုထဲမှာ ပါဝင်တဲ့ Unique Value အရေအတွက်ကို ရေတွက်ပါတယ်။ ဒီ Function ကို Threat Detection မှာ အရမ်းအသုံးများပါတယ်။

ဥပမာအားဖြင့် Password Spraying Attack ကို စစ်ဆေးချင်တယ်ဆိုပါစို့။ Password Spraying ဆိုတာက Attacker တစ်ယောက်က Password တစ်ခုတည်းကို User အကောင့် အများကြီးနဲ့ စမ်းသုံးတဲ့ တိုက်ခိုက်မှုမျိုး ဖြစ်ပါတယ်။ ဒီလိုအခြေအနေမှာ Logon Attempt တွေအားလုံးကို ကြည့်နေရုံနဲ့ မမြင်နိုင်ပါဘူး။ Aggregation လုပ်ပြီး Pattern ကို ကြည့်ရပါမယ်။

```
external_table('Winlog')
| where EventCode==4624
| summarize Count=dcount(TargetUserName) by IpAddress
| sort by Count desc
```

ဒီ Query မှာ EventCode 4624 ဖြစ်တဲ့ Logon Event တွေကိုပဲ ရွေးထားပါတယ်။ ထို့နောက် IpAddress တူတဲ့ Event တွေကို Group လုပ်ပြီး အဲဒီ IP Address တစ်ခုချင်းစီကနေ Logon လုပ်ထားတဲ့ TargetUserName အရေအတွက်ကို dcount() နဲ့ ရေတွက်ထားပါတယ်။

IpAddress	Count
192.168.56.110	4
-	4
127.0.0.1	2
192.168.58.170	2
192.168.56.111	2
192.168.58.167	1
192.168.56.102	1
192.168.58.1	1

Rows per page 10 1-8 of 8

Result ကို ကြည့်လိုက်ရင် User အကောင့် အများကြီးကို Logon လုပ်နေတဲ့ IP Address တွေကို အပေါ်ဆုံးမှာ တွေ့ရပါလိမ့်မယ်။ ဒီလို Pattern က Password Spraying ဖြစ်နိုင်တဲ့ လက္ခဏာတစ်ခု ဖြစ်ပါတယ်။

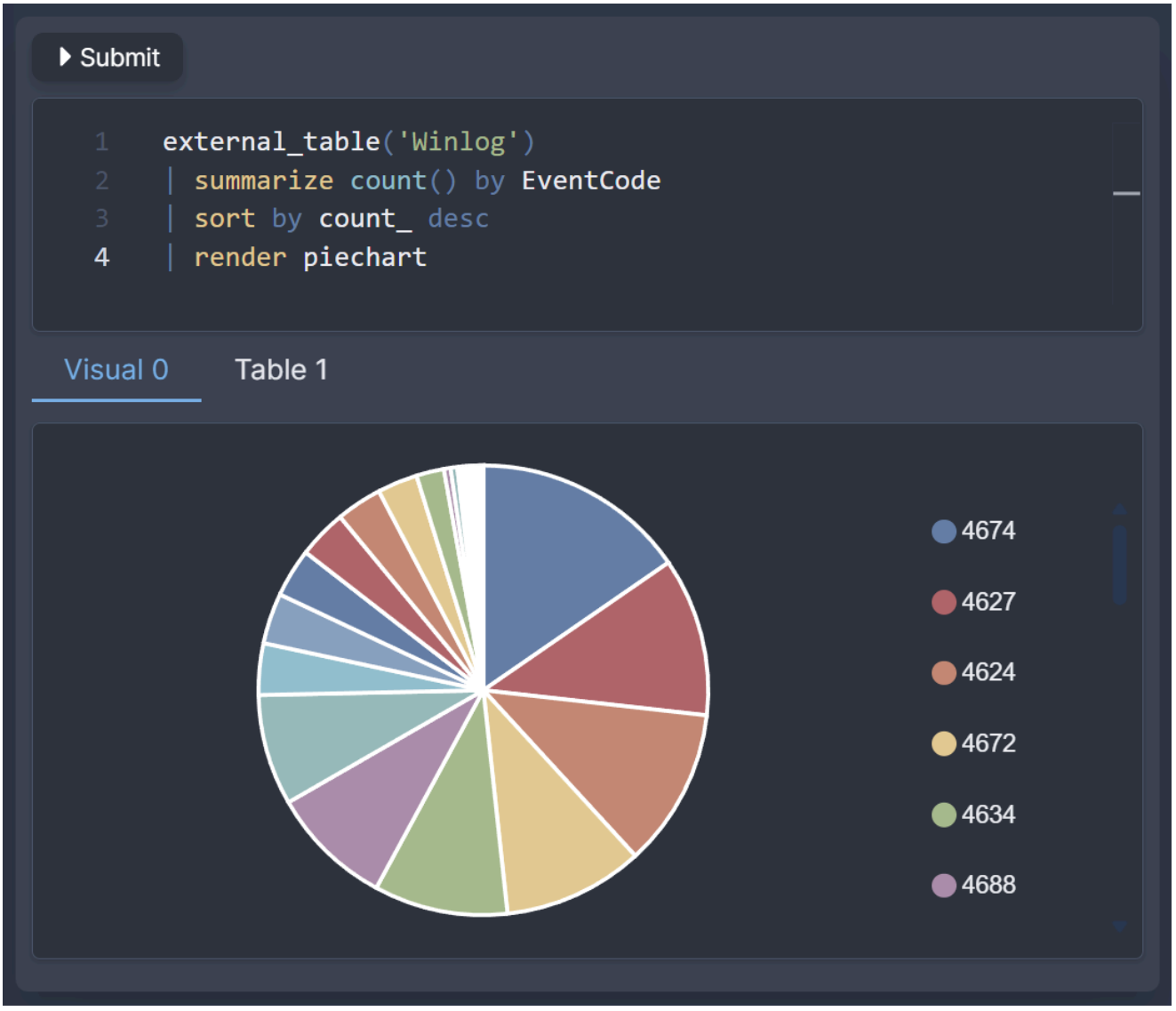
နေ့စဉ် SIEM Analysis အများစုမှာ count() နဲ့ dcount() ဆိုတဲ့ Function နှစ်ခုကိုပဲ အသုံးပြုရတာ များပါတယ်။ တချို့အခြေအနေတွေမှာတော့ sum() ကို အသုံးပြုပြီး Host တစ်ခုကနေ ထွက်လာတဲ့ Byte အရေအတွက်ကို တွက်တာလို အလုပ်တွေ လုပ်နိုင်ပါတယ်။ ဒါပေမယ့် နေ့စဉ် Threat Hunting နဲ့ Incident Triage အတွက်တော့ ရိုးရိုး Aggregation တွေကို မှန်ကန်စွာ အသုံးချနိုင်ခြင်းက အရေးကြီးဆုံး ဖြစ်ပါတယ်။

ဒီသဘောတရားတွေကို နားလည်ထားရင် SIEM ထဲမှာ Log Data အများကြီးကြားထဲကနေ Threat Pattern တွေကို မြန်မြန်နဲ့ တိတိကျကျ ဖော်ထုတ်နိုင်လာပါလိမ့်မယ်။

Visualizations

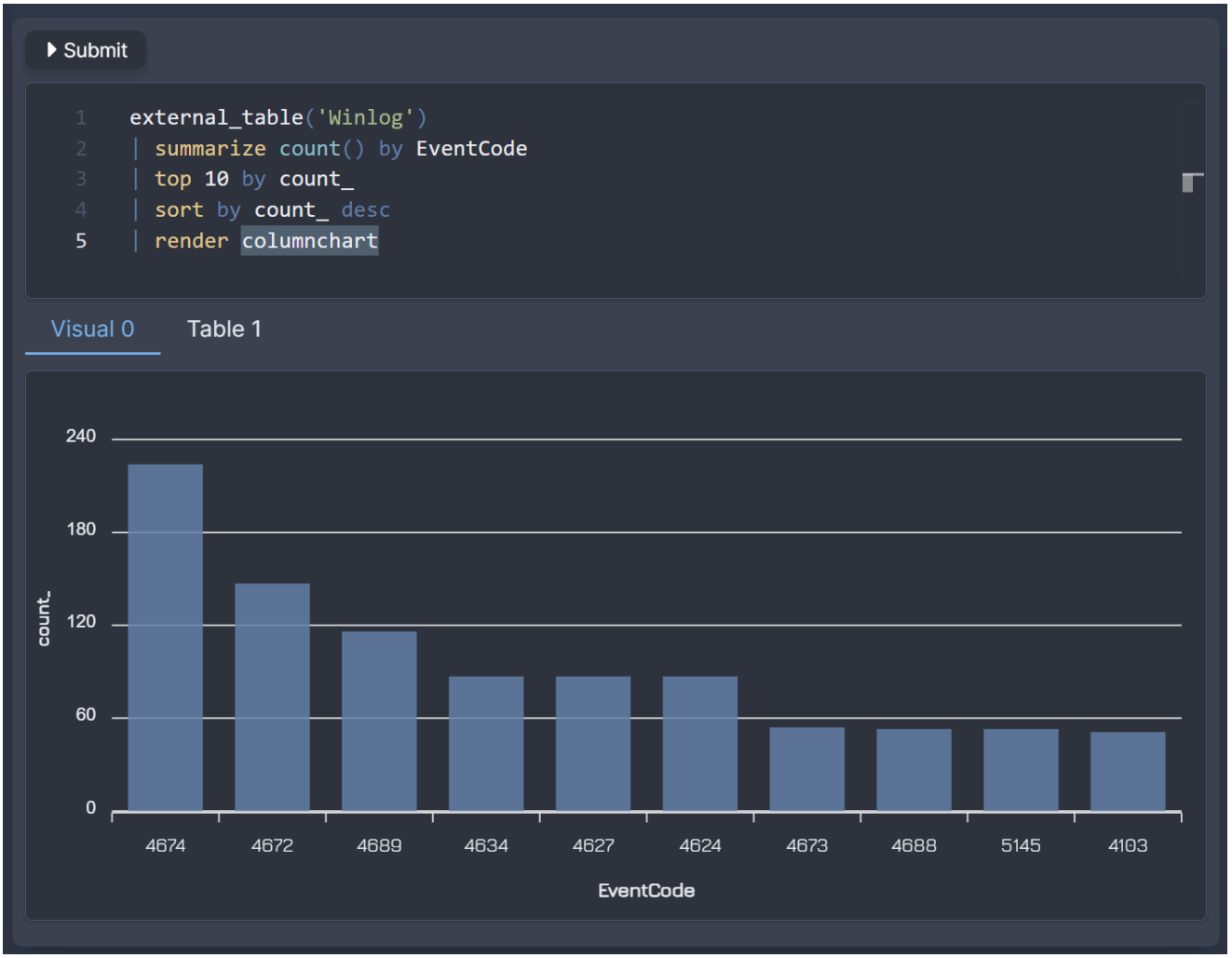
SIEM ထဲမှာ KQL Query တွေကို ရေးပြီး Result ကို Table အနေနဲ့ မြင်ရတာက အရေးကြီးပေမယ့် တစ်ခါတစ်ရံ Table ကို ကြည့်ရုံနဲ့ Pattern ကို ချက်ချင်း မမြင်နိုင်ပါဘူး။ အထူးသဖြင့် Data အများကြီး ပါလာတဲ့အခါ လူမျက်စိနဲ့ စာရင်းကိန်းတွေကို တစ်ခုချင်းစီ လိုက်ဖတ်နေရင် အချိန်ကုန်ပြီး အမှားလည်း ဖြစ်နိုင်ပါတယ်။ ဒီအချိန်မှာ Visualization ကို အသုံးပြုရင် Data ကို မြင်သာအောင် ပြောင်းလဲပြီး Threat Pattern တွေကို တစ်ချက်ကြည့်နဲ့ နားလည်နိုင်လာပါတယ်။

အရင်ဆုံး EventCode အလိုက် Count လုပ်ထားတဲ့ Query ကို ပြန်ကြည့်ရအောင်။ ဒီ Query က Database ထဲမှာ ဘယ် EventCode က ဘယ်လောက် အကြိမ် ဖြစ်နေတယ်ဆိုတာကို ပြပေးထားပြီးသား ဖြစ်ပါတယ်။ ဒီ Table ကို ကြည့်ရုံနဲ့လည်း အချက်အလက်တွေ ရရှိပေမယ့် Visualization လုပ်လိုက်ရင် ပိုပြီး အမြင်ရှင်းလာပါတယ်။



KQL မှာ Visualization လုပ်ဖို့ render operator ကို အသုံးပြုပါတယ်။ render ရဲ့ အလုပ်လုပ်ပုံက ရိုးရှင်းပါတယ်။ summarize နဲ့ Aggregation လုပ်ပြီးသား Result Table ကို Chart အနေနဲ့ ပြောင်းပြပေးတာပါ။ Query ရဲ့ အဆုံးမှာ render နဲ့ Chart အမျိုးအစားကို ထည့်ရေးလိုက်ရုံနဲ့ Kusto က အလိုအလျောက် Chart ကို ဆွဲပေးပါတယ်။

EventCode အလိုက် Count လုပ်ထားတဲ့ Data ကို Pie Chart အနေနဲ့ ပြချင်တယ်ဆိုရင် EventCode က အပိုင်းပိုင်း ခွဲထားတဲ့ Category ဖြစ်ပြီး count_ က အချိုးအစား ဖြစ်လာပါတယ်။ Pie Chart ကို အသုံးပြုရင် Event အမျိုးအစား တစ်ခုချင်းစီက စုစုပေါင်း Event ထဲမှာ ဘယ်လောက် ရာခိုင်နှုန်း ပါဝင်လဲဆိုတာကို တစ်ချက်ကြည့်နဲ့ နားလည်နိုင်ပါတယ်။ ဒီလို Chart က Environment ထဲမှာ ပုံမှန်ဖြစ်နေတဲ့ Event တွေနဲ့ ထူးခြားတဲ့ Event တွေကို ခွဲခြားဖို့ အကူအညီပေးပါတယ်။



Pie Chart အပြင် Column Chart ကိုလည်း အသုံးပြုနိုင်ပါတယ်။ Column Chart က EventCode တစ်ခုချင်းစီကို Column တစ်ခုစီနဲ့ ပြပေးပြီး Count ကို အမြင့်အနိမ့်နဲ့ ဖော်ပြပေးပါတယ်။ ဒီ Chart ကို အသုံးပြုတဲ့အခါ Result ကို Top 10 လောက်ပဲ ကန့်သတ်ထားရင် ကြည့်ရလွယ်ပြီး Chart လည်း မရှုပ်ထွေးပါဘူး။ Column Chart က Event အများဆုံး ဖြစ်နေတဲ့ Category ကို အလွယ်တကူ ဖော်ထုတ်နိုင်စေပါတယ်။

Timecharts

အခု Timecharts အကြောင်းကို ဆက်ပြီး နားလည်ကြည့်ရအောင်။ SOC Analyst တစ်ယောက်အနေနဲ့ Data ကို အချိန်အလိုက် ကြည့်ဖို့ အလွန်စိတ်ဝင်စားရပါတယ်။ ဥပမာအားဖြင့် Bob ဆိုတဲ့ User က Host အများကြီးကို Logon လုပ်နေတယ်ဆိုတာကို တွေ့ခဲ့ရင် အဲဒါက ပုံမှန်လား၊ မပုံမှန်လား ဆိုတာကို ချက်ချင်း မဆုံးဖြတ်နိုင်ပါဘူး။ ဒီအတွက် အခြား User တွေရဲ့ Activity နဲ့ နှိုင်းယှဉ်ပြီး အချိန်အလိုက် Pattern ကို ကြည့်ရပါမယ်။

ဒီလို Analysis အတွက် အကောင်းဆုံး Visualization က `Timechart` ဖြစ်ပါတယ်။

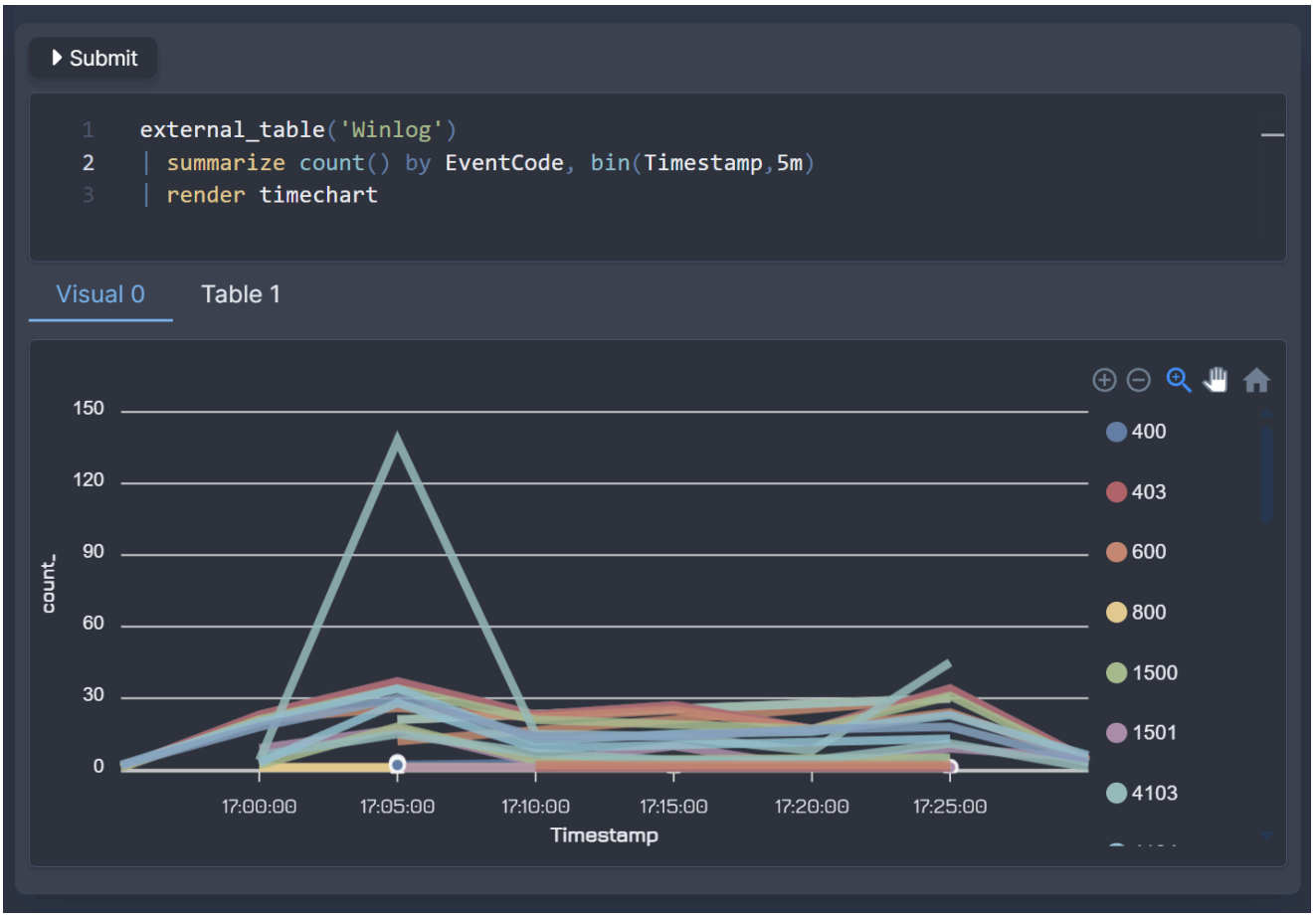
> Timechart ဆိုတာက အချိန်ကို X-Axis အနေနဲ့ ထားပြီး Event အရေအတွက်ကို Y-Axis အနေနဲ့ ပြတဲ့ Line Chart သို့မဟုတ် Area Chart ဖြစ်ပါတယ်။ ဒီ Chart ကို ကြည့်ရင် Activity က တဖြည်းဖြည်း တက်လာလား၊ တစ်ချက်တည်း ပေါက်ကွဲသလို မြင့်တက်လာလားဆိုတာကို မြင်နိုင်ပါတယ်။

EventCode Visualization ကို ဆက်ပြီး Time-based Visualization အဖြစ် ပြောင်းချင်ရင် summarize ကို နည်းနည်း ပိုပြီး တိုးချဲ့ရပါတယ်။ EventCode အလိုက်ပဲ Group မလုပ်ဘဲ အချိန်အလိုက်ပါ Group လုပ်ပေးရပါမယ်။ ဒီလိုလုပ်ခြင်းက EventCode တစ်ခုချင်းစီကို အချိန်အပိုင်းအခြားလိုက် ခွဲပြီး Count လုပ်နိုင်စေပါတယ်။

ဒါပေမယ့် Timestamp ကို တိုက်ရိုက် Group လုပ်လိုက်ရင် ပြဿနာတစ်ခု ဖြစ်ပါတယ်။ Timestamp ဟာ Millisecond အဆင့်အထိ တိကျတဲ့အတွက် Event တစ်ခုချင်းစီက ကိုယ်ပိုင် Timestamp ရှိပြီး Group မ ဖြစ်သွားပါဘူး။ ဒီလိုဖြစ်ရင် Chart က တစ်ပြားတည်း ပြားသွားပြီး အဓိပ္ပါယ် မရှိတော့ပါဘူး။

ဒီပြဿနာကို ဖြေရှင်းဖို့ `bin()` Function ကို အသုံးပြုပါတယ်။ bin() က Timestamp ကို သတ်မှတ်ထားတဲ့ အချိန်အပိုင်းအခြားအလိုက် Rounded လုပ်ပေးပါတယ်။ ဥပမာအားဖြင့် 17:31 နဲ့ 17:45 ဆိုတဲ့ Timestamp နှစ်ခုကို bin(Timestamp, 1h) နဲ့ ချိန်ညှိလိုက်ရင် နှစ်ခုစလုံးကို 17:00 အဖြစ် သတ်မှတ်ပြီး အဲဒီ တစ်နာရီအတွင်း ဖြစ်တဲ့ Event တွေကို တစ်စုတည်း Group လုပ်ပေးပါတယ်။

ဒီလို Time Slice လုပ်ခြင်းက မိနစ်အလိုက်၊ နာရီအလိုက်၊ နေ့အလိုက် Event Activity ကို ကြည့်နိုင်စေပါတယ်။ Result ကို Timechart အနေနဲ့ Render လိုက်ရင် Event Activity က အချိန်အလိုက် ဘယ်လို ပြောင်းလဲနေတယ်ဆိုတာကို တစ်ချက်ကြည့်နဲ့ နားလည်နိုင်ပါပြီ။ ဒီ Pattern ကို ကြည့်ပြီး ပုံမှန် Baseline နဲ့ မတူတဲ့ Spike တွေကို ချက်ချင်း သတိထားမိနိုင်ပါတယ်။



bin() ကို အသုံးပြုတဲ့အခါ Timespan ကို အမျိုးမျိုး သတ်မှတ်နိုင်ပါတယ်။ 1h ဆိုရင် တစ်နာရီအလိုက်၊ 1d ဆိုရင် တစ်နေ့အလိုက်၊ 30s ဆိုရင် စက္ကန့် ၃၀ အလိုက် Group လုပ်ပါတယ်။ ဒီ Shorthand Timespan Literal တွေက အရမ်း လွယ်ကူပြီး SIEM Analysis မှာ နေ့စဉ် အသုံးများပါတယ်။

အကျဉ်းချုပ်ပြောရရင် Visualization ဟာ Aggregation နဲ့ ပေါင်းပြီး SIEM Analysis ကို အဆင့်တစ်ဆင့် မြှင့်တင်ပေးပါတယ်။ Table ထဲမှာ မမြင်ရတဲ့ Threat Pattern တွေကို Chart အနေနဲ့ ပြောင်းကြည့်လိုက်ရင် SOC Analyst တစ်ယောက်အနေနဲ့ ပိုမြန်မြန် ဆုံးဖြတ်နိုင်ပြီး Incident Response ကို ပိုမို ထိရောက်စေပါလိမ့်မယ်။
